Dynamic Resource Management and Job Scheduling for High Performance Computing

Dynamisches Ressourcenmanagement und Job-Scheduling für das Hochleistungsrechnen

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Suraj Prabhakaran, M.Sc aus Vellore, Indien
Tag der Einreichung: 30.08.2016, Tag der Prüfung: 14.10.2016
Darmstadt 2016 — D 17

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr-Ing. André Brinkmann

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Laboratory for Parallel Programming

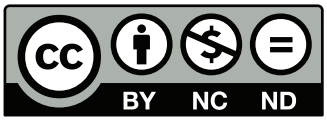
Dynamic Resource Management and Job Scheduling for High Performance Computing
Dynamisches Ressourcenmanagement und Job-Scheduling für das Hochleistungsrechnen

Genehmigte Dissertation von Suraj Prabhakaran, M.Sc aus Vellore, Indien

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr-Ing. André Brinkmann

Tag der Einreichung: 30.08.2016
Tag der Prüfung: 14.10.2016

Darmstadt 2016 — D 17



To
my parents

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30.08.2016

_____

(Suraj Prabhakaran)

Abstract

Job scheduling and resource management plays an essential role in high-performance computing. Supercomputing resources are usually managed by a batch system, which is responsible for the effective mapping of jobs onto resources (i.e., compute nodes). From the system perspective, a batch system must ensure high system utilization and throughput, while from the user perspective it must ensure fast response times and fairness when allocating resources across jobs.

Parallel jobs can be divided into four categories - rigid, moldable, malleable, and evolving. While rigid jobs have fixed resource requirements over their entire life cycle, moldable jobs allow batch systems to deviate from the requested number of resources before job start. In contrast, malleable and evolving jobs can adapt to changing resource allocations at runtime. While batch systems can expand or shrink a malleable job's resource allocation at any point of time, expanding and shrinking an evolving job occurs only in response to a request made by the application itself.

Traditional batch systems support only rigid and moldable jobs, that is, they perform static resource management. However, this is not sufficient as supercomputing enters a new era. Scientific applications are becoming much more complex and now often exhibit unpredictably changing resource requirements. Programming models are also becoming more adaptive in nature to support malleability for energy efficiency and fault tolerance. Therefore, scheduling evolving and malleable jobs (i.e., dynamic resource management) will be indispensable, especially on future large-scale systems. This dissertation therefore proposes novel dynamic resource management and scheduling techniques for cluster systems, making multiple contributions in the areas of dynamic resource (de)allocation mechanisms, efficient adaptive job scheduling, and resiliency.

As the first contribution, this thesis presents dynamic scheduling methods for evolving jobs. A fairness scheme is proposed to ensure the fair allocation of resources between static and dynamic resource requests. The evaluation with a workload containing both rigid and evolving jobs shows that high resource utilization and throughput can be achieved, while maintaining the fair dynamic assignment of resources. It is also demonstrated how these methods can be beneficially employed in heterogeneous architectures with network-attached accelerators.

The second contribution presents a unique scheduling technique for malleable jobs and an algorithm for the combined scheduling of all four types of jobs in a cluster environment. We introduce the Dependency-based Expand/Shrink (DBES) algorithm, which rests on a two-phase malleable job expand/shrink strategy. The batch system is evaluated with a mixed workload and our strategy achieves consistently superior performance in comparison to state-of-the-art malleable job scheduling strategies.

Finally, as the last contribution, we present a scheduling algorithm for dynamic node replacement, which improves the resiliency of cluster systems. The algorithm uses the unique features of the four job types and can provide replacement nodes instantly to jobs affected by node failures. Among current fault tolerance mechanisms, our technique causes the smallest loss of throughput.

Zusammenfassung

Job Scheduling und Ressourcen-Management spielen eine wesentliche Rolle im Bereich High-Performance Computing. Die Ressourcen von Hochleistungsrechnern werden für gewöhnlich von einem Batch System verwaltet, welches für die effektive Abbildung von Jobs auf Rechenknoten verantwortlich ist. Aus der Systemperspektive betrachtet, muss das Batch System für hohe Systemauslastung und Durchsatzleistung sorgen. Aus der Sicht des Nutzers sollte es eine gerechte Verteilung der Ressourcen und schnelle Antwortzeiten sicherstellen.

Es ist üblich, parallele Jobs in vier Klassen zu unterteilen - Rigid, Moldable, Malleable und Evolving. Während Jobs der Klasse Rigid festgelegte Vorgaben zur Auftragserteilung haben, erlauben Jobs der Klasse Moldable dem Batch System, die Zahl der Ressourcen vor der Ausführung zu verändern. Demgegenüber können sich Jobs der Klassen Malleable und Evolving auch zur Laufzeit an eine wechselnde Ressourcenzuteilung anpassen. Batch Systeme können die zugeteilten Ressourcen eines Jobs der Klasse Malleable zu jeder Zeit ausweiten oder verkleinern. Bei einem Job der Klasse Evolving kann dies jedoch nur auf ausdrückliche Anfrage der Anwendung selbst erfolgen.

Traditionelle Batch Systeme unterstützen nur Jobs der Klassen Rigid und Moldable (statische Ressourcenverwaltung). Dies ist allerdings angesichts der aktuellen Entwicklung nicht mehr ausreichend. Wissenschaftliche Anwendungen sind sehr viel komplexer geworden und weisen nun häufig unvorhersehbare Veränderungen ihres Ressourcenbedarfs auf. Zusätzlich werden Programmiermodelle anpassungsfähiger und unterstützen Malleability zur Verbesserung der Energieeffizienz und Fehlertoleranz in kommenden Exascale-Systemen. Daher wird auch das Scheduling von Jobs der Klassen Evolving und Malleable unverzichtbar werden. Aus diesem Grund präsentiert diese Dissertation neue Techniken zur dynamischen Ressourcenverwaltung und zum Scheduling auf Cluster Systemen und leistet dadurch Beiträge in den Bereichen dynamische Ressourcen(de)allokation, effizientes adaptives Jobscheduling und Resilienz.

Zunächst wird ein dynamisches Schedulingverfahren für Jobs der Klasse Evolving vorgestellt. Ein Fairness Konzept sorgt für die gerechte Verteilung von Ressourcen zwischen statischen und dynamischen Anfragen. Die Auswertung unter einer Last bestehend aus Jobs der Klassen Rigid und Evolving zeigt, dass unter der Wahrung einer fairen Ressourcenzuordnung eine hohe Systemauslastung und Durchsatzleistung erreicht werden kann. Es wird zudem gezeigt, wie diese Funktion in unkonventionellen, heterogenen Architekturen verwendet werden kann.

Der zweite Beitrag der Arbeit umfasst ein neuartiges Verfahren für Jobs der Klasse Malleable sowie einen Algorithmus für die gemeinsame Planung aller vier der oben genannten Jobklassen auf einem Clustersystem. Vorgestellt wird der DBES Algorithmus, welcher zwei Phasen zur Ausweitung und Verkleinerung von Malleable Jobs vorsieht. Das Batch System wird mit einer gemischten Last untersucht. Die Ergebnisse zeigen eine überlegene Leistung im Vergleich zu anderen modernen Schedulingverfahren.

Der dritte Beitrag ist schließlich ein Algorithmus zum dynamischen Austausch von Knoten, der geeignet ist, die Fehlertoleranz von Clustersystemen zu erhöhen. Der Algorithmus nutzt die spezifischen Eigenschaften aller vier Jobtypen und ist dadurch imstande, ausgefallene Knoten zu ersetzen. Unter den aktuellen Fehlertoleranz-Mechanismen bietet der vorgeschlagene Algorithmus den geringsten Durchsatzverlust.

Acknowledgements

When I started working on this PhD project, I did not think that it will change my life forever. Working on it for a little less than 5 years, it has been an eventful time filled with hard work, lots of learning, and unforgettable experiences. This would not have been possible without the support of many people towards whom I feel deeply grateful.

First and foremost, I would like to thank my advisor Prof. Dr. Felix Wolf for providing me a great platform with freedom to pursue my ideas and at the same time guiding every little project of mine with absolute involvement to its successful completion.

I would also like to thank all my colleagues at both TU Darmstadt and German Research School for Simulation Sciences who always created the best work environment and have been strong instruments of support at various points of time during this project. In particular, I want to thank Sebastian Rinke for the insightful discussions that always influenced my projects positively.

My gratitude goes out to everyone I have had the chance to collaborate and work with: Christian Windisch from the German Research School for Simulation Sciences, Prof. Laxmikant Kalé and Abhishek Gupta from University of Illinois at Urbana-Champaign, Gary Brown from Adaptive Computing, and Dong Ahn from Lawrence Livermore National Laboratory. I would also like to acknowledge the members of the DEEP project for providing me with the opportunity to work in a EU-funded project composed of a highly qualified team of partners. A special thanks to Adrian Spona from the German Research School for Simulation Sciences who provided me a cluster platform almost exclusively to evaluate my work.

Finally, my deepest gratitude to my family - my parents, my brother and my sister-in-law. They always bore the brunt of my frustrations during testing times and extended only unconditional support, for which I am greatly indebted.

Contents

List of Figures

List of Tables

1 Introduction

This chapter motivates the thesis, states its objectives and the contributions of this thesis. It provides a brief description of the context and the background necessary for understanding the problem from which the objectives are derived. Thereafter, the main contributions of this thesis are highlighted.

1.1 High Performance Computing

Scientific simulations have always been a powerful medium for research and development. In simple words, they can do what lab experiments cannot. Simulating physical and hypothetical phenomena has been monumental to a wide extent - from understanding the fundamental science to solving and predicting the outcome of sophisticated scenarios. For example, brain simulations aim to study and understand how information leading to well-known human behavior is encoded in the brain. Weather simulations predict the everyday change in the weather. Such simulations require large supercomputers to execute in a timely manner due to their demands of computational power and memory. Therefore, parallel programming has always been at the heart of these applications. With progress in science and improvements to underlying models for accuracy, simulations are growing more complex. Also, an increasing number of scientists from new domains are moving towards simulations with unconventional parallel programs. Thus, the computational power-hungry nature of scientific applications has been a driving force in the research and development of high performance computing architectures. Thus, high performance computing (HPC) can be simply defined as the use of parallel computing at large scale for faster problem solving. More precisely, it is use of supercomputers and parallel processing techniques to perform computations in acceptable time.

1.1.1 Hardware architectures

Since the emergence of the idea of combining multiple processors for computation, the size of supercomputers has been constantly growing. One of the earliest architecture along this idea was the shared-memory multiprocessor or symmetric multiprocessor (SMP). An SMP system consists of multiple *processing elements* sharing one large global memory. Today, this is reflected in multicore architectures where each processing element is a *core* of a processor chip and more than one processor are present in the same computer. Applications can use any number of cores and execute in parallel while sharing the same memory, which is naturally better than executing on uniprocessors for both commercial as well as HPC purposes. An example of a powerful multicore processor for HPC used today is the Blue Gene/Q processor [3] chip. It has 18 cores and provides 16 cores for computing while retaining one core for the operating system

services and one core as spare. Over the years, the chip technology has improved considerably to accommodate more and more cores in a processor chip. However, the scalability in these architectures is always limited by memory bandwidth.

Distributed-memory systems, also known as massively-parallel processors (MPP), overcome this problem. They consist of several computer systems with their own memory interconnected to perform computing tasks together. Today, each computer system or a node comprises one or more multicore processors. Although the network transfer of data comes as a bottleneck, MPP systems achieve very good scalability with fast interconnects such as Infiniband, EXTOLL [4] or other customized interconnects. Several organizations combine commodity processors with Infiniband interconnects and create small clusters to achieve parallel computing with reasonable performance for a specialized class of applications.

Many supercomputers deliver the high performance with an element of *heterogeneity*. A large number of cores for high end computing are found in *accelerators*, which are either *graphic processing units* (GPUs) or *coprocessors*. They are typically connected to a computer as a PCI Express card and consist of several cores and own memory.

Todays most powerful supercomputers are able to perform computations more than one quadrillion (10^{15}) FLOPS (floating point operations per second) or 1 petaFLOPS. The TOP500 list [5], which lists the 500 most powerful computing systems in the world, currently consists of 95 petascale systems (June 2016). The Sunway TaihuLight supercomputer at the National Supercomputing Center in Wuxi tops the list with a peak performance of 93 petaFLOPS. It consists of 40,960 nodes with each node consisting of one many-core processor chip called SW26010. Each SW26010 chip consists of 260 cores, thus making a total of 10,649,000 cores in the whole system. The Tianhe-2 supercomputer at the National Super Computer Center in Guangzhou stands second in the list with a peak performance of 33.8 petaFLOPS. It consists of 16000 nodes, each with two Intel Ivy Bridge Xeon processors and three Intel Xeon Phi coprocessors [6], thus having a total of 3,120,000 cores. And the third fastest supercomputer, Titan at Oak Ridge National Laboratory, delivers a peak performance of 17.5 petaflops with 18,688 nodes, each containing one 16-core AMD Opteron 6274 processor and one Tesla K20X GPU [7]. The next generation of supercomputers are the exascale systems, which will be capable of more than one quintillion (10^{18}) FLOPS, exceeding the capability of today's machines by a factor of little more than 10 (given that the top supercomputer today can already perform 93 quadrillion FLOPS).

1.1.2 Software stack

A number of software components and libraries are essential to correctly operate and use a supercomputer. And each supercomputer may be equipped with different software packages with varying levels depending upon on the architecture of the supercomputer. Figure 1.1 represents a generic HPC software stack with vital components and they are briefly described below.

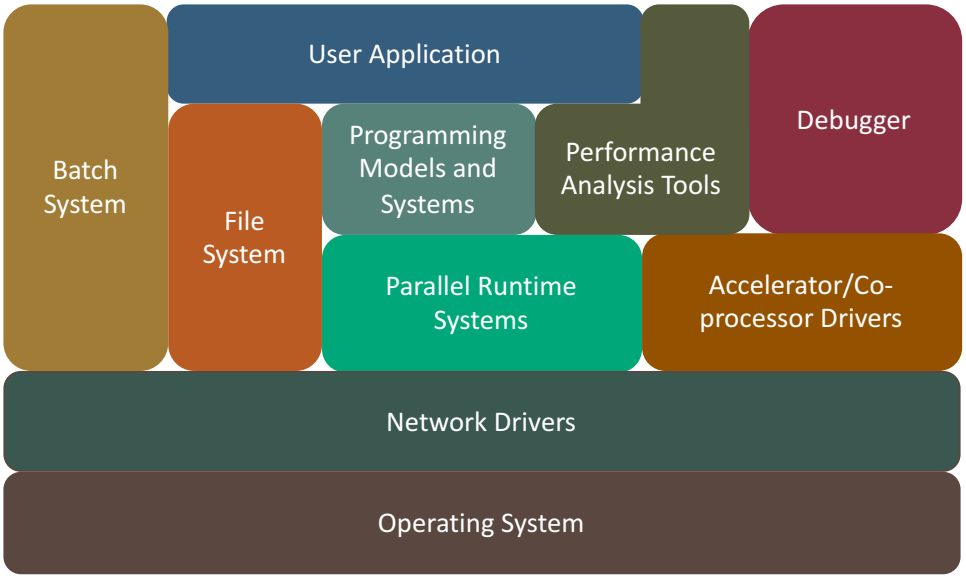


Figure 1.1: HPC software stack.

User Application. A user application is a program that is to be executed on the system and the results of which are of direct value to the user. An example is a weather simulation that predicts the everyday change in climate, as stated in Section 1.1.

Programming models and systems. The most important software package necessary for execution of a parallel application is an appropriate parallel programming model and programming system/paradigm. The user must select an appropriate model and paradigm for the application based on the architecture and the algorithm used. In distributed-memory systems, message passing between multiple processes executing an application in parallel has been the most popular method. In this respect, MPI [8] is the de-facto standard for message passing in HPC. Typically, MPI processes execute in parallel across the nodes of a cluster and can communicate between each other through various mechanisms such as point-to-point and collective communication. The processes of an MPI program belong to a *communicator*, which is an abstract context, and hold a unique identifier called *rank* under each communicator. MPI is among the most widely used parallel programming models found on HPC systems today.

In shared-memory systems, parallelism is realized mostly by employing multithreading. The POSIX threading interface (Pthreads) is a classic example of a library that allows multithreading in a convenient way. However, the OpenMP API [9] stands favorite for HPC applications. OpenMP uses a portable and scalable model that provides a simple and flexible interface to programmers. With multicore systems increasing drastically today, the importance of OpenMP has tremendously grown. Users also combine MPI and OpenMP (hybrid programming) to exploit parallelism at every level. Cluster-wide parallelism is achieved through MPI, while node-level parallelism is gained through OpenMP.

A programming paradigm that is suitable for both architectures is the Charm++ [10] parallel programming system. A Charm++ program consists of potentially medium-grained processes (called chares), a special type of a replicated process, and collections of these chares. These processes interact with each other via messages. The system can be considered a concur-

rent object-oriented system with a clear separation between sequential and parallel objects. Charm++ programs are written in C++ with a few library calls and an interface description language for publishing Charm++ objects. Charm++ supports multiple inheritance, late bindings, and polymorphism. Owing to its popular features such as efficient portability, dynamic load balancing, modularity and latency tolerance, Charm++ is used by many scientific applications from various domains [11, 12, 13, 14].

On the other hand, heterogeneous systems need special programming models. Using the GPUs efficiently requires the use of CUDA [15] or OpenCL [16]. CUDA (Compute Unified Device Architecture) is a programming model developed by NVidia for their own GPUs. OpenCL, however, is a hardware-independent standard for programming across heterogenous platforms.

Debuggers. Debuggers are handy tools that help programmers to test and identify bugs in the code. Given that parallel programs are more complex than serial programs, using debuggers is an efficient way of testing the code and can save a lot of time. GNU GDB [17] is an example of a popular debugger that can be used to debug serial and multi-threaded programs. However, debugging parallel programs involving parallel processes requires more comprehensive tools that can also display the information effectively to a user. Rogue Wave TotalView [18] and Allinea DDT [19] are examples of debuggers that can be used for analyzing MPI programs with an easy-to-use interface.

Performance analysis tools. Parallelism depends not only the appropriate programming model used but also on correctly exploiting the various aspects of the programming paradigm and the application algorithm. Owing to the complex nature of scientific applications, understanding how to fully harness the potential of a supercomputer is not easy. Performance analysis tools assist the programmer in identifying the performance bottlenecks. They can help in pinpointing many facets such as load imbalances and hotspots in communication and computations. This is usually achieved by first instrumenting the code and measuring various performance-relevant metrics. The measured data is then stored and subject to analysis along with with an acceptable way of presentation to the user. Different techniques can be employed at each stage depending upon the tool and the user's requirements. Prominent examples of some advanced tools are TAU [20], HPCToolKit [21] and Scalasca [22, 23].

Parallel runtime systems. The parallel runtime system is the engine that implements the execution model underlying a parallel programming model. For example, MPICH [24] and Open MPI [25] are prominent examples of MPI runtime systems used today. Depending upon the programming models supported by a runtime system, it becomes responsible for managing several functions such as establishing communication between the processes of a parallel program, scheduling work, and load balancing. Parallel runtime systems can also be used to build further runtime systems to support various programming models. For example, Adaptive MPI (AMPI) [26] implements the MPI standard on top of the Charm++ runtime system, which primarily enables the execution model of the Charm++ parallel programming language.

Accelerator and co-processor drivers. Heterogenous systems that contain accelerators and/or co-processors require special drivers to be installed on each node, without which they will not be able to be used. The drivers also provide an API that can either be used by a runtime system or directly by the application. For example, using NVIDIA GPUs requires CUDA drivers, which are used by the CUDA runtime. Applications can also use the CUDA driver API to perform computations on the GPU.

File system. File systems are an important part of a HPC system. An HPC system executes several jobs in a day, which may potentially have many data-intensive applications. Therefore, HPC systems always offer a parallel file system which can be used by applications to store data persistently and use them during execution. BeeGFS [27] and Lustre [28] are some notable parallel file systems used in today's cluster systems. Applications are given access to the file systems using a file system API. For example BeeGFS and Lustre can be accessed through the POSIX interface or MPI I/O.

Batch system. One of the key components for a cluster's operation is the *batch system*. A batch system manages the HPC resources for all the applications in the form of *jobs*. Its main functionality includes:

1. Submit jobs - Provide an interface for users to submit jobs and queue them.
2. Manage job execution - Execute a job under the requested environment and allow users and administrators to control it.
3. Job scheduling - Map the jobs from the queue onto the resources available in the cluster.

A batch system is also important for application performance, as the choice of resource allocation can influence its execution time (e.g., allocating physically closer nodes to a job can reduce latency and allow faster communication). Thus, it is responsible for the complete operation of a cluster through resource management and job scheduling for users and administrators. Batch systems are also known in other terminology as resource and job management systems (RJMS), resource management systems (RMS), and batch job scheduler. The contribution of this thesis is towards advancements in resource management and job scheduling through batch systems. The following sections elaborate on batch system operation and introduce the problem addressed by this thesis.

Network drivers. All of the components above use network drivers to communicate with the other nodes of a cluster. HPC systems typically contain different network devices such as Ethernet and Infiniband. They require their drivers to be installed and usable by runtime systems to be able to perform network communication. For example, the OFED stack [29] provides drivers to perform many standard protocols such as IP over Infiniband.

Operating systems. The functions of an operating system (OS) is preferred to be minimal in a HPC environment. This is because, HPC systems aim to provide as many resources as possible to the user application and application libraries rather than to a resource-hungry kernel

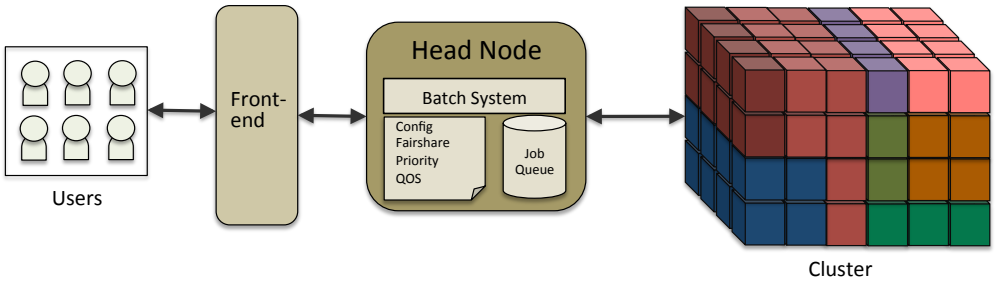


Figure 1.2: A typical setup of a cluster system. The head node runs the batch system. Users login on the frontend to submit and control jobs.

that implements full features of an OS. The Linux OS distributions such as RHEL [30] and CentOS [31] are some of the common choices of HPC system providers.

1.2 Managing HPC Resources - Background

This section provides an overview of the HPC resource management and job scheduling through batch systems. It introduces the functioning of batch systems and presents the classification of parallel jobs in a cluster environment. It then describes the aspects through which batch systems are assessed and evaluated.

1.2.1 Job scheduling and resource management

A batch system consists of a *resource manager* and a *job scheduler*, which work together to run jobs in a cluster environment. The resource manager is the component responsible for accepting jobs from the users and managing their execution on the nodes. It carries the task of communicating with each node, spawning process on the nodes and terminating them when necessary. It ensures that the required software environment is provided to the job for executing the application. The job scheduler is responsible for mapping the the jobs with the resources. Typically a scheduler first processes the job queue and the list of resources available. Then it designates resources for the jobs according to various policies and factors. The resource manager then takes care of executing the job on the nodes allocated for it. Schedulers today are built with competent algorithms for prioritizing jobs in the queue and performing the best allocation according to the needs of a site.

Figure 1.2 illustrates a typical setup of a cluster environment. Clusters usually provide a *frontend* where users can login and submit their jobs by specifying resource requirements (e.g., number of nodes, type of nodes). The batch system runs on a *headnode* to which jobs are submitted. Each cluster node runs a *control daemon* to which the instructions of starting and controlling a are communicated from the batch system at the headnode.

In addition to just executing jobs, a batch system is responsible for ensuring efficient operation of the cluster. From the system perspective, it must maintain a high system utilization and throughput so as to deliver a high availability of the system. From the user perspective, it must ensure a fair allocation of resources across jobs and deliver reduced waiting times of jobs in the

queue. Above all, the batch system must be configurable for allocation policies according to a site's preferences. For example, certain sites may require fairness between the users as the most important consideration when allocating nodes to jobs, while other cites may prefer to target high system utilization and throughput at the expense of fairness.

1.2.2 Parallel job classification

As defined by Feitelson and Rudolph [32], parallel jobs can be classified in four different categories based on the flexibility of resource usage.

Rigid jobs. A rigid job is the most common type of job found in today's cluster environments. A rigid job is submitted to a batch system requesting a fixed number of nodes necessary to execute the parallel program. The number of nodes is unmodifiable after job submission and throughout its execution. Most parallel applications are rigid in nature due to various reasons, the most common being the algorithm used in the application itself, which can only be executed on a fixed number of nodes. For example, the decomposition of a given problem size of an application may only be suitable for a specific number of nodes. A rigid job also defines an execution time for the application which it is not allowed to cross. There are many methods for scheduling rigid jobs with the aim of improving throughput and response times such as best-fit algorithms and backfilling. Backfilling is the process of scheduling jobs out of order from a FIFO queue as long as those jobs do not delay the start time of a configurable number of jobs placed higher up in the queue.

Moldable jobs. A moldable job is similar to a rigid job except that the batch system is allowed to change its resource requirements after job submission but before job start. The batch system can choose to execute the job with smaller or greater number of resources than the main resource requirement specified, for example, to map the job onto the idle nodes currently available and improve performance. Therefore, moldable jobs are submitted by specifying a range of number of nodes acceptable for the job and a definite execution time for each specification. MPI jobs are moldable as long as the application is able to decompose the problem according to the number of resources available for execution.

Evolving jobs. Contrary to rigid and moldable jobs, an evolving job can change its resource allocation set (or reservation) during job execution. An evolving job initiates a change in resource allocation by dynamically requesting additional nodes from the batch system. Based on this request, the batch system *expands* the job. At the same time, the application can also dynamically release some of its nodes and reduce the size of its reservation, which is termed job *shrink*. A job can evolve due to various reasons. For example, the application may have incurred additional computations as an outcome of intermediate results which require additional resources to be able to finish the job within the specified walltime limit. In some cases, applications may not be able to continue execution without additional resources (e.g., when the application reaches hardware limits such as memory). In such scenarios, the application

requires additional resources to distribute the data and computations across them and continue execution.

Malleable jobs. Malleable jobs are the most scheduler-friendly jobs. The batch system can expand or shrink a malleable job at any point of time. The application will adapt itself to the changed resource set. The most common programming model that allows malleability is OpenMP. As more cores of a node are made available to a process, the parallelism can be altered transparently between the parallel sections of the program. However, since production clusters allocate nodes exclusively to a job, there is no practical usage for it. Other adaptive programming models like Charm++, AMPI [26] and OmpSS [33] allow multi-node malleability. There are various ways in which the resource requirements can be specified for a malleable job. The most common method is by specifying a minimum, an ideal, and a maximum number of nodes required by the job so that the job's allocation can be expanded or shrunk within the specified range. The batch system can flexibly change the resource set of a malleable job to achieve a good system utilization and throughput. This also benefits the user in multiple ways. For example, a malleable job can be started as soon as the minimum number of nodes required are ready and expanded later as more nodes become available.

Since the allocation of resources for rigid and moldable jobs are made before job start and cannot be changed thereafter, it is termed as *static allocation*. The process of expanding or shrinking a resource allocation of a malleable or evolving job (called together *adaptive* jobs) is termed *dynamic allocation*.

1.2.3 Performance aspects

In the context of resource management and scheduling, the *system performance* is measured through the metrics described below.

System throughput. System throughput is the most important metric to measure the effectiveness of a batch scheduler. It is defined as the number of jobs completed per unit time. Ideally, the throughput must be as high as possible as it leads to better availability of the system. Since a cluster typically receives a mix of long and short running jobs, the scheduler must ensure an effective execution of jobs so as to maintain good throughput and avoid jobs from experiencing resource starvation.

System utilization. As the name indicates, system utilization is a measure of the amount of resources being used by jobs. It is usually represented as a percentage of resources used over a time frame by periodically observing their usage. The scheduler must aim at maintaining high system utilization to avoid resource wastage. Resource wastage not only leads to low throughput but also increases the costs for system providers. On the other hand, high system utilization does not always mean high throughput. An ineffective job and resource mapping can also lead to high system utilization without delivering the best throughput. Therefore, a good scheduler must aim both at high system utilization as well as high throughput.

Makespan. Makespan is the time taken for the completion of a fixed workload. Makespan is a common and convenient metric to evaluate and compare different scheduling algorithms. A lower makespan value implies higher throughput and therefore better scheduling. However, it is unsuitable for determining the quality of scheduling in a production machine as they do not have fixed workloads and jobs often arrive at irregular intervals.

Waiting time. Waiting time is the amount of time a job spends in the queue waiting to be executed. That is, it is the difference between the time at which the job starts and the time at which it was submitted to the batch system. For a given workload, the average waiting time can be used to evaluate the effectiveness of a scheduler. A lower average waiting time indicates better scheduling. The waiting time is most suitable for evaluating fairness strategies. In general, a fairness policy with equal priority for all users aims at maintaining a comparable average waiting time of jobs among the users.

Job turnaround time. The turnaround time refers to the total time between job submission and job completion. In other words, it is the time spent by the job waiting in the queue plus the duration of its execution. The average turnaround time (e.g., for a user, a workload, a group of users) is often used when evaluating a scheduler. The batch system can influence the turnaround time of a job only by reducing its waiting time. The duration of execution depends on the hardware and the application. However, for the other job types, the batch system can also influence the execution time. Providing a higher number of nodes/cores requested by a moldable job usually leads to faster execution. Expanding malleable jobs may reduce an application's execution time. Similarly, the batch system can also help evolving jobs complete on time or even earlier by satisfying their dynamic resource allocation requests. Therefore, job turnaround time can be used to evaluate both static and dynamic allocation schemes.

Response time. The formal definition of response time is the difference between the time at which the job was submitted and the time at which the first response or output was received from the batch system. This is synonymous to turnaround time for batch jobs as the first output received for a batch job from the batch system is only after the job completion. It differs from turnaround time only for interactive jobs. Since most jobs executed in HPC clusters are batch jobs, researchers have used this metric synonymously with job turnaround time. As this thesis also deals only with batch jobs, turnaround time and response time have been used interchangeably.

Fairness. Fairness is more of a quality than a metric and hence there is no single metric that can be used to measure it. At the same time, as the number of users of HPC clusters have rapidly increased, it stands as the most important property that a batch system must establish. One of the most common ways of measuring it is by comparing the average waiting time of jobs across multiple users. An approximately similar average waiting time usually indicates fair scheduling. On the other hand, when users have an unequal mix of job types, the average turnaround time of each user's set of jobs could be dissimilar. Typically, reputed batch systems use a mix of these metrics to ensure fairness across users. Above all, since fairness is also a political issue,

it is established according to the needs of a site by allowing administrators to specify various parameters such as priorities of a certain users or groups of users.

The ideal scheduler is naturally one that provides the best of all the above. However, job scheduling for clusters is an online scheduling process. That is, the state of a queue changes incessantly with jobs being added or removed and therefore, the best possible schedule cannot be projected beforehand. In general, maintaining high system utilization and throughput (often under a site-specific fairness policy) is a universal goal that leads to the best performance with respect to other metrics as well. The performance of a batch system depends not only on the scheduler but also on the workload. With a larger presence of moldable and malleable jobs, the batch system can flexibly allocate resources to jobs to achieve good performance even with a rapidly changing job queue.

1.2.4 Batch system evaluation

Batch systems are mainly evaluated for their efficiency in scheduling and resource allocation than for the resource management overheads such as communicating with cluster nodes and spawning processes on remote nodes. This is because the resource management overheads are usually negligible in comparison to the running time of scientific applications. There are two principal ways of evaluating a batch system: (i) through real experiments, and (ii) through simulations.

Real experiments. The natural way of evaluating a batch system is by deploying it in a real cluster environment and analyzing its effectiveness in scheduling a workload. However, doing so is not an easy task. Replacing the production batch systems with custom ones for the purpose of evaluation endangers the management of a large number of jobs submitted by users. Also, installing a batch system requires administrator privileges, which is reserved only for designated system administrators and not granted to other users. Therefore, sites usually do not allow experimental batch systems to be installed. While there are some large-scale clusters that allow batch system evaluation (such as the Grid5000 [34] cluster hosted by INRIA), most production batch systems are evaluated with in-house small-scale clusters.

Even with a suitable cluster environment, performing real experiments is a cumbersome task. Firstly, it requires the implementation of the scheduling strategies directly in production batch systems, which are typically complex software packages with large amount of code. The programming effort needed for such an implementation is momentous. Secondly, debugging and testing can only be performed with small experiments which requires repeated installations of the batch system (including the control daemons) after every modification. Large experiments are prone to unexpected errors, which may increase the need for restarting the experiments multiple times. However, though it is a difficult and time consuming process, it is the most reliable way of evaluation.

Simulations. Simulation is one of the convenient ways of evaluating multiple aspects of resource allocation and scheduling. Many researchers evaluate scheduling algorithms with

custom-developed simulators that can simulate a cluster system and the execution of a queue of jobs in that system [35],[36]. Wholesome cluster simulators with diverse features and deep levels of detail are also available. A well-known simulator is the Structural Simulation Toolkit (SST) [37] developed at Sandia National Laboratories. SST consists of a parallel simulation core with a number of network, memory and processor models capable of evaluating systems at different levels of resolution. It also consists of high-level system models of scheduling and node allocation, which can be used to analyze various scheduling algorithms. Similar to SST, GridSim [38] and CloudSim [39] are popular simulators for simulating Grid and Cloud environments respectively. They also consist of node allocation modules to evaluate scheduling strategies. SimGrid [40], on the other hand, provides functionality to simulate a wide variety of systems. It consists of an SMPI component [41] which enables the detailed simulation of individual MPI jobs.

Another class of simulators are those dedicated to the evaluation of scheduling and processor allocation techniques. The Parallel Resource Management Algorithm Simulator (PReMAS) [42] uses a simple model for jobs and nodes and reports the quality of processor allocation or task mapping in terms of specific metrics. ProcSimity [43] also allows the analysis of scheduling algorithms under various metrics along with the ability to model message flits moving through the network.

Such simulators ease the process of evaluation as they greatly reduce the coding effort which is needed when having to perform real experiments with production batch systems. However, simulations cannot be used to determine the scalability and the overhead of resource management. They also pose only limited possibility to analyze scheduling effects under a number of real conditions such as site-specific policies, sudden failures of hardwares, dynamic expansion of the cluster partition and administrator control actions.

In both cases, the evaluation can only be performed with a reasonable workload of jobs. Since system performance depends both on batch scheduling and workload characteristics, it is important to evaluate batch systems with a workload that puts its distinct features to test. This is usually achieved in two ways by using: (i) a workload with the traces of real workloads in popular clusters, and (ii) benchmark workloads

Real workload traces. Real workload traces from well-known cluster systems are a popular choice when testing resource allocation strategies. A repository of workload information on parallel machines is freely available in the Parallel Workloads Archive [44]. It contains raw workload logs from various machines around the world and workload models that are derived from the logs. The workload logs are represented in the Standard Workload Format (SWF) [45], which lists all the essential information of a job such as its submit time, requested duration and actual run time. For the evaluation, dummy jobs emulating the characteristics of those in the trace are used. This method is widely accepted since imitating a real workload directly enables studying the performance of a job scheduling strategy under realistic conditions and patterns of cluster usage. On the other hand, a drawback of this approach is that a real workload may not

necessarily test a scheduler with an exhaustive set of scheduling scenarios. An evaluation with multiple real workloads may be necessary to study all aspects of scheduling.

Benchmark workloads. Using benchmark workloads and workload models (called *synthetic workloads*) is also a reliable way of evaluating resource allocation strategies. Benchmark workloads are normally created by analyzing multiple real workloads and particularly ordered to pose challenging scenarios to a scheduler. The workload models available in the Parallel Workloads Archive can be used for this purpose. The most prominent benchmark workload used frequently by researchers is the ESP benchmark [46]. It consists of a throughput workload with varying partition sizes and times, which is inspired from a large number of scientific applications [46]. The main objective of the ESP test is to run a fixed number of parallel jobs through a batch system with minimum makespan. The ESP benchmark has been and still continues to be used regularly for evaluating scheduling algorithms [47],[48],[49].

It is important to note that all the workloads (both real and benchmark) available today use only rigid jobs. While there are few real workloads with moldable jobs, evolving and malleable jobs are completely absent. This is because batch systems have supported only rigid and moldable jobs over the years.

1.3 Motivation and Scope of this Thesis

Traditionally, most batch systems support only static allocations, the reason primarily attributed to the rigid nature of the majority of parallel applications, which in-turn is due to the less adaptivity-friendly nature of commonly used programming models like MPI. Although many applications written with MPI are moldable, only a few batch systems such as SLURM [50] and Moab Workload Manager [51] support scheduling moldable jobs. To write evolving applications with MPI, application developers have to use the MPI-2 dynamic process management facilities [52] to manually manage processes on a new set of resources. Processes have to be spawned on the new hosts using the `MPI_Comm_spawn()` call and the resulting intercommunicator must be changed to a intra-communicator. Thereafter, users must also manage the data distribution manually across all the processes. Therefore, users desired writing more static applications that did not require programming effort for managing processes and resources. Most importantly, the application domains explored by scientists over the years did not have evolving characteristics. Similarly, writing malleable applications with MPI is a daunting task. Since the batch system can expand or shrink a job at any point of time, the application must be ready to adapt itself to the changing resource set. As MPI does not allow communicators to shrink or expand, they must be made flexible through the MPI-2 dynamic process management features, which requires a huge amount of programming effort. Although there has been some work on easing the process of writing malleable MPI applications [53] [54], the applicability of these techniques are limited to only a few application domains and are hardly found in today's cluster systems. Thus, dynamic resource-management facilities were not strongly required so far. This scenario has been changing due to multiple reasons.

The first is the demand of applications. As scientists explore new application domains, scientific simulations are showing strongly evolving behavior. The most common scenario causing job evolution is unanticipated intermediate results leading to increase in computations. Typical examples are applications using adaptive mesh refinement [55] or multiscale analysis [56] such as Quadflow [57]. These applications consist of a grid or a mesh, which has a fixed number of cells at the beginning, but may change during the course of execution in the grid adaptation phase. The computations of the application increases as the grid grows and more cells are produced. For the simulation of certain real-world problems, a rapid growth of the grid can lead to the program reaching the memory limit on a node. In such a case, the application even cannot continue execution without additional resources. As the growth of cells depend entirely on the intermediate results produced by the computations, it cannot be predicted before job start for most problems. Another class of applications showing evolving behavior are those that require the execution of smaller secondary simulations alongside the main simulation. For example, weather simulations require the simultaneous execution of nested simulations to keep track of multiple weather phenomena [58]. Extra resources are needed only when executing the additional simulations. Similarly, brain simulations also require additional resources to execute analysis phases at various points of time during the execution of the main simulation.

The second reason is the change of nature of the programming models from being rigid to becoming more adaptive. Programming models are primarily trying to meet the needs of fault tolerance, load imbalance and energy efficiency for upcoming exascale systems. Enabling these features requires applications to adapt to a changing resource set. For example, the Charm++ runtime can autonomically manage resources that are allocated to a Charm++ job, identify and ameliorate load imbalance, adapt the application to a changing resource set, as well as cope with the intermittent loss of resources due to component failures. OmpSS also provides automatic load balancing and malleability of a job. A fault-tolerant version of MPI slated to be released in the near future to fit exascale systems will also bring adaptiveness to its runtime system [59]. Programming paradigms that support adaptivity are foreseen to play a significant role in exascale systems [60] [61]. Applications using these paradigms are automatically malleable and hold a strong potential to obtain high system performance. Using such programming paradigms paves the way for power-aware adaptive scheduling which has the ability to handle the energy challenge for future systems [61]. These paradigms become the first choice for writing evolving simulations as programmers do not have to manage the dynamically allocated resources manually. However, due to the lack of support for dynamic resource-management facilities in current batch systems, neither is the potential held by malleable jobs utilized nor are evolving jobs supported in today's cluster systems.

A third situation entailing dynamic resource management is the need for enabling better fault tolerance in upcoming exascale systems. The basic expectation of a user from a cluster environment is a correct and timely completion of jobs. For that matter, it is fundamental that supercomputers provide a robust and reliable environment. However, job interruptions due to both hardware and software failures will be one of the main roadblocks delaying or even preventing job completion at exascale. Exascale systems are expected to consist of billions of cores, which will also increase their overall failure rate [60]. In current petascale systems, the

mean time between node failures (MTBF) is in the order of hours. For example, the MTBF (single or multiple node failures) of the Blue Waters system has been 6.7 hours in an operation period of about 261 days in 2013 [62]. The MTBF of an exascale system is expected to shrink to only one hour or less. Frequent failures can have adverse effects on applications as they can interrupt long-running jobs multiple times.

The current strategy of handling job interruption includes user-initiated periodic checkpointing of the application and resubmission of the job into the batch system to restart the job with a fresh allocation of resources from its latest checkpoint. However, the reallocation process consumes non-negligible overhead and there may well not be enough nodes to re-launch the job immediately. One way of circumventing this problem is to allocate dedicated spare nodes alongside each job, beyond what is required by the application, so that these spare nodes can be put to use immediately in the event of a node failure. The application can be killed and restarted from its latest checkpoint without having to be resubmitted to the job queue. Unfortunately, in this approach, each job will require a relatively large number of spare nodes to stay prepared for faults. This leads to a significant amount of resources to remain unused and results in poor system utilization and throughput, reducing the overall availability of the system. Furthermore, a seemingly sufficient number of spare nodes allocated for each job may still turn out to be inadequate, as the pattern of faults may drastically vary. Thus, the job has to recline to requesting a fresh allocation through resubmission. Hence, such a static resource allocation mechanism cannot be effectively used for fault tolerance at exascale.

Finally, the progress towards exascale has encouraged exploring unorthodox architectures with network-attached accelerators. The DEEP cluster system [63] is a prominent example. It consists of a *cluster-booster architecture* where a cluster of accelerators called *booster* is available for any compute node over a high-speed network instead of the traditional way of attaching accelerators directly to a compute node through PCIe. The accelerators can be flexibly assigned to applications running in the compute nodes based on its demands at varying phases of execution, which is achievable only through dynamic allocation techniques.

Overall, dynamic resource management is an essential feature for future systems (i) to support the needs of increasingly complex applications, (ii) to improve the overall system performance, (iii) to enable advanced fault tolerance mechanisms, and (iv) support heterogeneous architectures with network-attached accelerators. Thus, the main goal of this thesis is to provide novel dynamic resource management and scheduling methods that is suitable for all the above purposes and can be integrated in production batch systems.

1.4 Contributions of this Thesis

While the contributions of this thesis are multifold, they can be summed up as a batch system with dynamic resource management and job scheduling techniques for scheduling adaptive jobs and enabling resiliency in cluster systems. The techniques are implemented in the well-known TORQUE/Maui batch system [64, 65] and a custom-discrete event simulator based on the GridSim [38] simulator. Each contribution made by this thesis is summarized below:

Support for evolving jobs. The first contribution of this thesis are the dynamic resource management and scheduling techniques for supporting evolving jobs in cluster environments [66]. We analyze the different types of evolving jobs and consider unpredictably evolving jobs for building dynamic resource management methods. We propose methods for processing evolving requests by considering the characteristics of unpredictably evolving applications, which are derived from the study of a real-world application that can unpredictably evolve. We show that scheduling resource requests from unpredictably evolving jobs can adversely affect the fairness between dynamic and static requests, which may potentially lead to resource starvation for jobs waiting in the queue. Therefore, we also propose a new fairness scheme to ensure the fair allocation of resources between dynamic and static requests. The scheme is incorporated into the scheduling of evolving jobs and can be configured according to a site's needs. Evaluation of an enhanced TORQUE/Maui batch system with a modified ESP benchmark that also consists of evolving jobs, shows that better system utilization, throughput and reduced waiting times can be achieved while keeping the overhead of dynamic allocation negligible.

Support for malleable jobs. The second contribution of this thesis is dynamic resource management and scheduling techniques for malleable jobs [67]. We show how malleable jobs can be supported in a cluster by tightly coupling the batch system and the parallel runtime. The Charm++ parallel runtime automatically makes applications malleable. The TORQUE/-Maui batch system is extended with dynamic resource management features to expand and shrink a malleable job and is tightly coupled to the Charm++ parallel runtime with an API to pass shrink/expand messages. We propose a novel algorithm for scheduling the expansion and shrinkage of malleable jobs called Dependency-Based Expand/Shrink (DBES). Evaluation of the algorithm (implemented in TORQUE/Maui batch system) with a modified ESP benchmark that also consists of malleable and evolving jobs shows that it performs consistently superior to every other malleable job scheduling strategy for varying dynamics of the workload. Moreover, the strategy is combined with the scheduling of evolving jobs. The evaluation with a workload mix of rigid, malleable and evolving jobs shows that combining these strategies provides a better throughput and system utilization over other methods. Finally, the shrink/expand overhead is also shown to be negligible.

Improved fault tolerance with dynamic node replacement. The third contribution of this thesis is enabling improved fault tolerance through dynamic resource management and node replacement. This feature enables the batch system to dynamically replace failed nodes of a job when it is affected by hardware failures. This avoids the necessity to restart the job on a fresh allocation and can potentially eliminate the need for disked checkpointing. We present a novel scheduling algorithm for dynamic node replacements that leverages the unique features of all four job types. The algorithm selects the best way of replacing nodes for a failed job with the aim of reducing the decrease in throughput that is inevitably caused by a hardware failure. Implementation and evaluation of the above are with a custom discrete-event simulator based on the GridSim simulator, which shows that dynamic node replacement has multiple advantages as compared to the traditional approach.

Applicability of dynamic resource management in heterogeneous architectures with network-attached accelerators. Finally, we show the applicability of our dynamic resource management methods developed in this thesis to heterogeneous architectures with network-attached accelerators [68]. We show how dynamic resource management methods are particularly relevant in architectures with network-attached accelerators by presenting their applicability to the DEEP cluster system [63] and the Dynamic Accelerator-Cluster (DAC) Architecture [2]. The DAC architecture realizes network-attached accelerators by using nodes connected with GPUs as accelerators and enables computational offloading on remote GPUs. Network-attached accelerators can be dynamically (de)allocated to a job based on its accelerator requirements at different computational phases. We show that flexible accelerator allocation can be achieved with negligible overhead.

1.5 Structure of this Document

This document is structured as follows. We start by discussing related work in the field of resource management and scheduling for evolving, malleable, and moldable jobs in Chapter 2. Also, existing fault tolerance techniques which only use static allocation schemes are reviewed. Chapter 3 provides an overview of the TORQUE/Maui batch system on top of which most of the methods developed in this thesis are implemented. Thereafter, we present each contribution of this thesis in a separate chapter. Chapter 4 and Chapter 5 describes our contribution for scheduling evolving and malleable jobs, respectively. Both chapters present an overview of the scheduling approaches for such jobs and then describe in detail the implementation, the scheduling algorithms and their evaluation. Chapter 6 presents the dynamic node replacement facilities for fault tolerance, the scheduling algorithm, and its evaluation. The applicability of the dynamic resource management features to heterogeneous architectures with network-attached accelerators is discussed in Chatper 7. Finally, Chapter 8 presents the conclusion of this thesis and an outlook on future research.

2 Related Work

This section reviews the previous research related to the contributions of this thesis. More precisely, it presents the relevant past work on dynamic resource management and scheduling of evolving, malleable and moldable jobs. Thereafter, fault-tolerance support of existing batch systems are discussed.

2.1 Evolving Jobs

Efficient resource management and scheduling of rigid jobs on cluster systems is a well studied topic that has seen substantial advancement. The growing complexity of applications and their adaptive nature has motivated many researchers to seek dynamic resource management and scheduling solutions. At the same time, many have also expressed that providing support for evolving jobs is a challenging task [69] [70]. Most of the work done in this filed is theoretical or based on simulation.

One of the early works in scheduling evolving jobs was performed by Boon-Ping and Shell-Ying [71] where dynamic scheduling is based on genetic algorithms. The approach was evaluated with simulators. However, whether the approach can be used with complex scheduling aspects such as prioritization, backfilling and fairshare, was not addressed. Investigating the RMS requirements for evolving jobs, Ghafoor et al. [72] proposed protocols for supporting evolving jobs and implemented a prototypical RMS to analyze the dynamic allocation overhead. The problem of scheduling evolving jobs was not considered.

The challenge of scheduling unpredictably evolving jobs was investigated by Klein et al. with the CooRMv2 RMS [73]. In their approach, along with the general job requirements, the maximum number of resources that may be dynamically required during execution must be indicated at job submission. These additional nodes are preallocated for the job and may be used only by malleable or preemptive jobs. Preemptive jobs are lower-priority jobs that can be cancelled when running and re-queued in order to make nodes available for other higher-priority jobs. The authors evaluate their approach with a workload of evolving and malleable jobs and prove the benefits. However, the applicability of preallocation method is limited to only a certain class of evolving jobs and can lead to drastic performance degradation in the absence of malleable or preemptive jobs. This is discussed in detail in Section 4.1.

Support for dynamic allocation on demand can also be found in the Moab Workload Manager [51] and the SLURM resource manager [50]. Moab supports resource expansion and shrinkage for evolving jobs by regularly querying each application about its load. However, this is available for interactive workloads only. The SLURM resource manager supports expand/shrink operations by allowing a running job to submit a new job with a *dependency* indicator and then merging the allocations. By submitting a new job, the existing static fairshare mechanism is used to prioritize the dynamic request. In this work, however, we distinguish

the dynamic and static requests and introduce new fairness schemes for scheduling. Moreover, SLURM's design demands that all the resources that were assigned for an evolving request be released together during a dynamic deallocation. Our approach provides more flexibility without any such a restriction. Jobs may dynamically deallocate any subset of their current allocation.

From a different angle, the fairness problem was addressed by Dinesh Kumar et al. [74] where jobs expand their walltime limit rather than consuming more resources. Their approach is based on extensions to a lookahead optimizing scheduler (LOS), which finds the best combination of jobs to be run simultaneously with the highest resource utilization. However, using the approach for dynamic allocations is not feasible.

The methods proposed in this thesis enable efficient dynamic allocation as well as effective dynamic fairness strategies which, to our knowledge, have not been studied before at the depth presented.

2.2 Malleable Jobs

The advantages of malleable jobs were theoretically identified several years ago. For this reason, frameworks for writing malleable applications have existed for more than a decade [75] [76]. Kale et. al. [77] developed an adaptive runtime system for Charm++ and showed the benefits of malleable jobs compared to rigid ones with an experimental scheduler using an equipartitioning policy, which distributes the idle resources to malleable jobs equally.

Since then, efficient resource management and scheduling for malleable jobs have been studied actively. Most of the work in this field pertains to theoretical aspects of scheduling and evaluation with simulations. For example, Carrol et. al [35] proposed a method for online scheduling of malleable jobs where the main goal was to assign resources to jobs such that the total running time of the application is reduced. Users submit a job along with an indication of the amount of time that will be required by the job to run on a single processor. To ensure that users do not manipulate the scheduler by misreporting the job's parameters, incentives were given to users if their job was completed on the specified deadline. Sun et. al. [36] proposed a scheduling strategy whereby resources are distributed to malleable jobs using the equipartitioning technique but periodically adjusted based on application feedback on its scaling pattern. Similar approaches were taken by Mounie et al. [78] and Blazewicz et al. [79].

Below, we discuss notable prototypical/demo schedulers for malleable jobs. Their scheduling methodology follows a standard approach: when the queued job with the next highest priority cannot be started anymore due to the lack of resources, the scheduler attempts to find nodes for the job by shrinking already expanded malleable jobs. When the next queued job cannot make use of any resources even by shrinking other jobs, then an expand phase is started where available idle resources are distributed across the running malleable jobs to improve system utilization and throughput. The order of jobs selected for shrinking and expansion varies according to the policy. Hungershöfer [80] showed that moldable and malleable jobs can significantly improve the response times through the equipartitioning strategy for shrink and expand. Utrera et. al. [81] proposed an FCFS-malleable strategy which distributes available nodes to malleable jobs in earliest-started-job-first order. They also investigated other strategies such as earliest

deadline first, latest deadline first and the one with the least CPU utilization first. They showed that for a cluster composed only of malleable jobs, the earliest started first strategy generally performed better and improved the average response time by 31% in comparison to well-known EASY backfilling. The OAR resource manager [82] was also extended to support malleable MPI jobs. However, the problem of scheduling multiple malleable jobs was not discussed.

In the context of grids, Buisson et. al. [83] introduced malleability in the KOALA multicluster grid scheduler with an equigrow and equishrink policy, a different flavor of equipartitioning. While the equipartition policy tries to equalize the amount of malleable nodes held by each running malleable job, the equigrow policy simply distributes the current set of idle resources equally among the malleable jobs. Thus, irrespective of the number of nodes held malleably by a job, it will be expanded by an equal proportion of idle nodes when the scheduler starts an expansion phase. This was combined with two scheduling approaches called as precedence to running applications (PRA) and precedence to waiting applications (PWA). PRA tries to expand as many running malleable jobs as possible, thereby prioritizing the allocation of idle nodes to running malleable jobs over queued jobs. PWA on the other hand attempts to allocate all the queued jobs before considering the expansion of running malleable jobs.

In general, naive equipartitioning was often employed and benefits were shown through prototypical schedulers. In this thesis, we propose dynamic scheduling methods for malleable jobs implemented in a production batch system. We show its benefits with Charm++ applications that become automatically malleable when run under the proposed batch system. We also evaluate a new malleable job scheduling strategy and compare it with naive equipartitioning, earliest started first, earliest deadline first and latest deadline first.

2.3 Moldable Jobs

Since enabling support for moldability is not as technically challenging as supporting evolving and malleable jobs, almost all current batch systems support moldable job scheduling. Platform LSF [84], OAR [82], SLURM and Moab HPC Suite [51] are some examples. Many methods for moldable job scheduling have been proposed in the last two decades, mostly aimed at improving the average turnaround time.

One of the early works on scheduling moldable jobs was proposed by Cirne et. al. [85]. They introduced a moldable application scheduler called Supercomputer AppLeS (SA) with the aim of delivering the best turnaround time possible. It analyzes each request by simulating the submissions made to the system and estimates the turnaround time for each job using a resource allocation list and a submit-time greedy strategy based on the current system state. The scheduler then allocates processors to jobs that are expected to deliver the least turnaround time. Srinivasan et. al. [86]observed that a greedy scheme can lead to unfair resource allocation for small jobs and therefore proposed a fair share strategy. This was further enhanced to be robust under different load and scalability conditions of jobs through extended reservation policies that limit partition sizes of jobs [87]. Gerald et. al. [88] proposed an iterative method for processor allocation, where decisions are made based on fair share allocation, overbooking, and job efficiency computed using the Downey model [89].

Moldable job scheduling has also been explored in various forms such as frameworks with schedulers and runtime for SAT solvers [90] and cloud environments [91]. Recently, Wu et. al. [92] proposed a scheme called HRF (highest revenue first), which allocates processors according to the highest revenue of shortening a job's runtime.

2.4 Fault Tolerance

Although fault tolerance and resiliency have been an active area of research for a long time, they have gained practical momentum in recent years as HPC progresses to exascale. Exascale systems are predicted to have high failure rates, which is motivating development of many methods for enabling a robust and reliable cluster environment.

In a broad perspective, failures can be categorized into two types: software and hardware failures. Software failures include the sudden death of important middleware such as the central batch system, control daemons on nodes, and runtimes and libraries used by the application. Batch systems typically save minimal state information of the cluster in file systems. This ensures that a batch system can be restarted without losing the information on the state of the jobs.

Fault tolerance against hardware failures has multiple aspects associated with it: fault detection, resource monitoring, and application recovery. Current production batch systems are already equipped with effective tools for resource monitoring and fault detection. These tools are also constantly improved for faster detection and low overhead monitoring. However, application recovery is a challenging aspect. The most popular approach for application recovery is through checkpoint/restart. The application is checkpointed regularly on disk storage and restarted from the latest checkpoint in the event it gets affected by a hardware failure. Some of the commonly used checkpoint/restart frameworks include BLCR [93], FTI [94] and SCR [95]. Since current batch systems only support static allocations, the saved job is first removed from the existing allocation and restarted on a fresh allocation. Batch systems such as Platform LSF, OAR, Moab HPC Suite, and HTCondor [96] follow this method. Also, since most batch systems do not support job types other than rigid jobs, it often causes failed jobs to wait for considerable time before a fresh allocation can be provided. The SLURM resource manager consists of a prototypical implementation of a node replacement facility. However, since it does not support malleable or evolving jobs, the waiting time for a node replacement is not improved compared to providing a fresh allocation.

Research in fault tolerance and resiliency had mostly had the focus on improved application recovery techniques such as enhanced checkpoint/restart methods, transparent and proactive process migration, process replication and algorithm-based fault tolerance. Programming models too have focussed on enabling a flexible and fault tolerant parallel runtime system. Batch systems are usually coupled with checkpoint/restart frameworks and tools that perform process replication and migration to ensure transparent recovery [97] [98]. In this thesis, we explore a new dimension on fault tolerance through dynamic node replacement which has not been deeply studied before. Dynamic resource management and node replacement open many opportunities for taking fast corrective responses, which are discussed in detail in Chapter 6.

3 The TORQUE/Maui Batch System

This chapter introduces the TORQUE/Maui batch system, which is used to implement majority of the methods proposed in this thesis. A general overview, the functionality and the capabilities of the TORQUE resource manager and the Maui scheduler are presented.

3.1 Job and Resource Management with the TORQUE/Maui Batch System

The TORQUE/Maui batch system is one of the most commonly used middleware systems for batch job control. The TORQUE resource manager [99] is based on the PBS project [100], extended to improve scalability and fault tolerance and is currently maintained by Adaptive Computing [101]. It is mainly responsible for accepting parallel jobs from users, executing and terminating jobs, job logging, and managing nodes. It is an open-source product and can be freely used, modified and distributed under the license from Adaptive Computing [64]. TORQUE also consists of a basic scheduler that can map resources and jobs based on the FIFO strategy. However, this scheduler is usually unused. TORQUE is more commonly integrated with sophisticated schedulers such as Maui [65], which provides advanced scheduling features. The widespread use of the TORQUE/Maui batch system was one of the principal reasons for choosing this ensemble to implement the dynamic resource-management facilities.

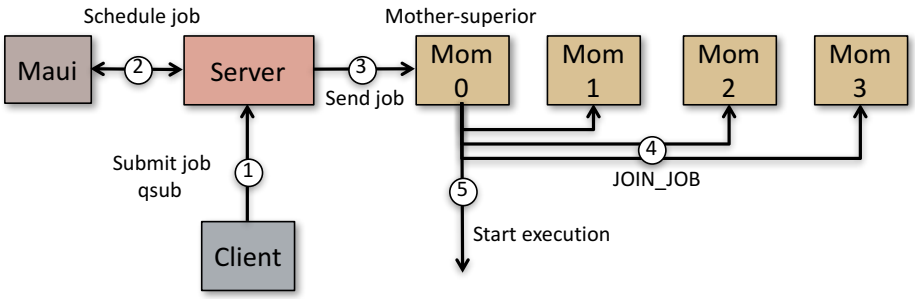


Figure 3.1: Workflow of the TORQUE/Maui batch system. Circled numbers indicate the sequence of steps.

A TORQUE/Maui cluster consists of a headnode, a frontend, and many compute nodes. The headnode runs the `pbs_server` daemon (`server`) and the Maui scheduler daemon. The compute nodes run the `pbs_mom` daemon (`mom`). Users are provided with a number of client commands to communicate with the `server` for tasks such as job submission, alteration and checking the status of a job. The client commands are usually installed on the frontend, but they can also be installed on any host including the headnode. The typical workflow of job scheduling in a TORQUE/Maui cluster is enumerated below and illustrated in Figure 3.1.

1. Users submit jobs with the `qsub` command indicating the required number of nodes (k), the cores per node (q), the duration of execution (x, known as *walltime*) and the script to be executed ($jobscript.sh$).

```
$ qsub -l nodes=k:ppn=q,walltime=x jobscript.sh
```

2. The `server` takes the request and stores the job information such as job submission time, resources required, etc. internally as *job attributes*. The job is then enqueued for resource allocation.

3. The Maui scheduler retrieves the list of queued jobs and the status of all the resources in the cluster from the `server`. It allocates resources to the jobs based on various site-specific policies and sends the information about the allocated resources for each queued job to the `server`.

4. The `server` reads the list of allocated hosts for a job and selects the `mom` running in one of the allocated hosts as the *mother-superior* and sends it the complete job information (including the list of allocated hosts).

5. The mother-superior, on receiving the instruction to run a job, first connects with each `mom` running on the other allocated hosts. The connection is established by sending what is called a `JOIN_JOB` request along with the job information.

6. After successful connection with all the `mom`s, the mother-superior starts the execution of the job script.

It is also common to install different type of resources in a cluster. For instance, a cluster may consist of a partition of nodes that have GPUs installed on them. In some cases, clusters have different type of GPUs installed in distinct node partitions. TORQUE allows requesting these resources separately by specifying the *type* of nodes required as shown below.

```
$ qsub -l nodes=2:general+4:gpunode jobscript.sh
```

In the above example, a user who requires 2 nodes without GPUs and 4 nodes with GPUS requests 2 nodes of type `general` and 4 nodes of type `gpunode`. Administrators can set the types for all nodes of the cluster in the configuration file. The keywords used as node type must be provided to the users so that they can request them appropriately. Also, a node can have multiple types associated with it. A job submitted to TORQUE goes through many *states*. After submission, the job enters the state of `waiting` or `queued`, which means that the job is waiting for a resource allocation from the scheduler. When resources are allocated and job execution begins, the state is changed to `running`. When a job is in the process of exiting (due to a a delete request from the scheduler, administrator or the user), it is changed to `exiting`. After the output has been written and the job has released all the nodes, it is changed to `completed`. When a job is running, the scheduler can suspend or hold the job for various purposes (e.g., to take checkpoints of the job or to kill the job preemptively and execute a higher priority job). At this stage, the state of the job is changed to `suspended`. A user can

identify the state of any of his/her jobs by using the `qstat` command, which returns the state of all of the user's jobs in the resource manager's queue. Similarly, TORQUE provides many other client commands to obtain and modify job information. For instance, the `qalter` command lets users change the job information depending upon the state of the job. For example, a user can change the resource requirements of a submitted job using the `qalter` command as long as it is in the `queued` state. Once the job has started running, such a request will be rejected by the resource manager. Similarly, a user can delete his/her job using the `qdel` command as long as the job is not already in the `exiting` or `completed` state.

Once the execution of a job begins, it can retrieve many information about its TORQUE environment by querying various environment variables set by the `mom`. For example, `PBS_JOBID` contains the global job-ID set by the `server` for this job. Typically, MPI jobs look into the `PBS_NODEFILE` environment variable which points to a file containing the list of hosts allocated for this job. All the processes of a job can communicate with their local `mom` through the **TM** API. An MPI runtime usually communicates with the local `mom` through the TM API to spawn processes on the remote nodes and subsequently control them. TORQUE also provides an IFL (Interface Library) API with which running jobs can communicate directly with the `server`. For example, `pbs_alterjob()` is equivalent to the `qalter` command and can be used from inside the application to alter certain properties of the job (e.g., walltime limit). The `pbs_statjob()` API call is useful to identify the status of another job belonging to the same user.

The complete list of features, a description of the API and other functionality provided by TORQUE are explained in the documentation available at the Adaptive Computing website [64].

3.2 The Maui Scheduler

The Maui cluster scheduler is an open-source job scheduler for HPC systems. It provides advanced features such as dynamic job prioritization, configurable parameters, extensive fairshare capabilities and backfill scheduling. The Maui scheduler functions in an iterative manner. In each iteration, Maui communicates with the `server` and then schedules jobs on the available resources with the consideration of diverse policies. A scheduling iteration is followed by a period of sleeping or processing external commands. Maui will instantly start a new iteration when (i) a job or resource state change occurs, (ii) a reservation boundary event occurs, (iii) an external command to resume scheduling is issued or (iv) a configurable timer expires. The steps of a scheduling iteration are detailed in Algorithm 1.

In each iteration, Maui obtains the most recent information about the jobs and nodes from TORQUE and updates this information internally (steps 1 and 2). It then updates the historical statistics of the jobs and the nodes (step 3). This is an important step as the fairshare policies use the historical usage data to prioritize jobs. In the next step, Maui refreshes all the previous reservations. In this step, jobs that were scheduled to start at that point will be started. All other reservations made for the queued jobs for a future point are cleared. This is because, in each iteration, Maui will schedule and create reservations for jobs after revising the latest circumstances of the system. After clearing the reservations, queued jobs meeting a

Algorithm 1 Maui iteration

1: **while** TRUE **do**
2: Obtain resource information from TORQUE
3: Obtain workload information from TORQUE
4: Update statistics
5: Refresh reservations
6: Select jobs eligible for priority scheduling
7: Prioritize eligible jobs
8: Schedule the jobs in priority order and create reservations
9: Backfill jobs
10: **end while**

minimum scheduling criterion based on throttling policies and job states are selected and considered again for scheduling (step 5). The selected jobs are prioritized according to various policies and then scheduled in the order their priorities. When a lack of resources prevents the next idle job with the highest priority from starting, the earliest time when the resources are available for this job is determined and a reservation is created (step 7). Maui continues to create reservations for N such highest priority jobs where N can be set by the administrator using the `ReservationDepth` parameter in the Maui configuration file. Jobs for which no reservation has been made are then backfilled out of order.

Backfilling is a strategy that allows the scheduler to improve resource utilization by putting the scheduling the queued jobs out of order. Since Maui has reservations for N jobs, it knows the earliest time at which the needed resources for these jobs will be available. Using this information, Maui determines which low priority jobs can be executed on the resources that are currently idle so that future reservations are not disturbed. That is, Maui determines which low priority jobs can be safely executed without delaying any of the reservations made to the highest priority jobs and executes them out of order. This ensures that the highest priority jobs are not delayed unfairly due to the execution of low priority jobs. At the same time, this technique improves the overall cluster utilization and throughput to a large extent. Note that reservations for queued jobs are made only in order to determine jobs for backfilling. When backfilling is disabled, reservations for jobs that cannot be started are not made. In practice, a large number of jobs in many cluster environments are small with respect to both the amount of resources and the duration of the job requested. These jobs can be easily backfilled.

However, one of the drawbacks of backfilling is that the resources left idle after running many large jobs may be scattered in the cluster. So short jobs that are backfilled are not likely to get nodes that are physically close to each other. Another problem is that backfilling can lead to resource starvation for the higher-priority jobs. If some jobs are backfilled and a running job terminates earlier than the walltime limit indicated (e.g., because of inaccurate walltime limit declaration by the user or an error in the application), this can delay the higher priority jobs since they now have to wait for the already backfilled jobs to complete to obtain enough resources for execution. This problem can be handled via preemption based backfilling. That is, when a scenario occurs where a higher priority job has to wait for the completion of a backfilled job with lower priority, the backfilled job can be killed and the resources can be taken away by

the higher priority job. The backfilled job that was cancelled is re-queued and executed later either when being backfilled again or when being pushed to the top of the queue. While this ensures that the high-priority jobs are not delayed beyond their original reservation, cancelling backfilled job wastes compute time, which the user cannot be charged for. Therefore, system administrators can enable or disable backfilling in accordance with site-specific policies. Also, the amount of jobs that should be backfilled can be controlled by the `ReservationDepth` parameter. A higher `ReservationDepth` leads to more conservative backfilling while a lower `ReservationDepth` allows more jobs to be backfilled.

The Maui scheduler allows setting different cluster usage policies for following entities, which are named political classes or credentials according to the Maui admin manual.

- **User** - An individual user of the cluster.
- **Group** - A group of users. For example, the permanent employees and temporary employees of an institution can be considered as two separate groups.
- **Account** - An account for the cluster based on payment for the cluster usage. For example, this can be the account of an institution available for all users belonging to it.
- **QOS** - Quality of service that must be provided for a certain user, group, or account. For example, users of group A can preempt jobs of users from group B.
- **Class** - The queue where jobs are submitted. Jobs are usually queued in the default queue. But separate queues or classes can be set for various users, groups, or accounts.

Maui computes the priority as a function (i.e., a waited sum) of the five main components listed below. Each of the component and the subcomponents that they consider can be associated with *weights* that influence the final priority of the job.

1. **Job credentials**
   The credentials of a job are directly defined by the political properties associated with it. Jobs can be priortized based on the associated user, group, account, QOS and the class that it belongs to. Administrators can set distinct weights for these subcomponents and control the priority relative to other jobs. A higher weight leads to a higher priority.
2. **Requested job resources**
   Jobs can also be prioritized based on the requested resources. This allows favoring jobs that can potentially increase the system utilization. Weights can be assigned to various properties such as the number of nodes requested, the type of nodes and amount of memory. Through this policy, administrators can increase the priority jobs with large resource demands and push the smaller jobs to the back of the queue so that they can be backfilled.
3. **Current service levels**
   The service component allows service-related factors to be included into the prioritization. For example, a job from a low-priority user can gain a higher priority based on the amount of time it has been waiting in the queue. Similarly, a job can be given higher priority when it has been bypassed too many times by backfilled low-priority jobs that caused it to starve.

4. **Target service levels**
   While the current service component schedules based on the job's current service levels, the target component allows scheduling in order to meet a particular target service.

5. **Fairshare**
   Fairshare is one of the most important components for scheduling. It allows favoring jobs based on political criteria for *fair* usage of resources across users, groups, accounts, QOS and classes. That is, system providers can define the fairness for all the political classes as a value that needs to be equal across them. Fairshare influences the prioritization of jobs based on short term historical usage of the cluster of the various political classes. For example, user B who submits a job half an hour after user A may still get a higher priority than user A because the latter has used distinguishably larger amounts of compute time in the past 24 hours than user B. Each of the credentials can be associated with a system utilization fairshare target. The fairshare usage is computed based on the historical usage and the duration of this interval can be set by the administrators. Jobs of users that have fairshare usage above the specified target will get a lower priority. Jobs that have not received enough resources during the specified interval and thereby have a fairshare usage lower than the target will get higher priority.

A detailed description of the components, prioritization factors and the exact method of computation is available in the Maui administrator manual [102].

4 Supporting Evolving Jobs

This chapter describes the dynamic resource management and job scheduling techniques developed in this thesis for supporting evolving jobs in a cluster environment. The chapter presents an overview of evolving applications and briefly describes Quadflow as a motivating example of evolving applications. Thereafter, methods for dynamic resource management and the scheduling algorithm designed are detailed. This is followed by the evaluation of the proposed methods with the application Quadflow and a modified ESP benchmark. The chapter concludes with remarks on the results and scope for future research.

4.1 Evolving Applications

As described in Section 1.2.2, evolving jobs have changing resource requirements during job execution. An evolving job can both request additional resources as well as release unused resources during runtime. As stated already, jobs that can potentially be evolving have an increasing presence in today's cluster environments due to the widespread use of techniques such as Adaptive Mesh Refinement (AMR) [55], Adaptive Particle Refinement (APR) [103] and multiscale analysis [56]. However, due to the absence of dynamic resource management capabilities, users of today's cluster systems execute these jobs as rigid ones.

4.1.1 Quadflow as a motivating example

The CFD flow solver Quadflow [57] is a classic example of an evolving application. It solves the compressible Navier-Stokes equations using a cell-centered fully adaptive finite volume method [104] on locally refined grids. Grid adaptation is based on multiscale analysis instead of classical gradient- or residual-based error estimators. The computational grids are represented by block-structured parametric B-Spline patches [105] to deal with complex geometries. In order to reduce the computational load to an acceptable amount, these tools are equipped with parallelization techniques based on space-filling curves [106] to run the simulations on distributed memory architectures. Starting on the coarsest grid level, the computational grid of the investigated flow configuration is successively refined until the final grid level is reached. The local refinement of the grid leads to high computational efficiency. However, since the areas in need of refinement can only be identified during the solution process, no a-priori knowledge is available on the development of the number of grid cells. This depends on a multitude of factors that govern the computations of different problems. Thus, as the number of grid cells increases, more computations are produced. Only for a small range of problems, the growth of cells and the iteration at which this may possibly happen can be predicted beforehand. Such a pattern can also be observed in several AMR applications [107, 108].

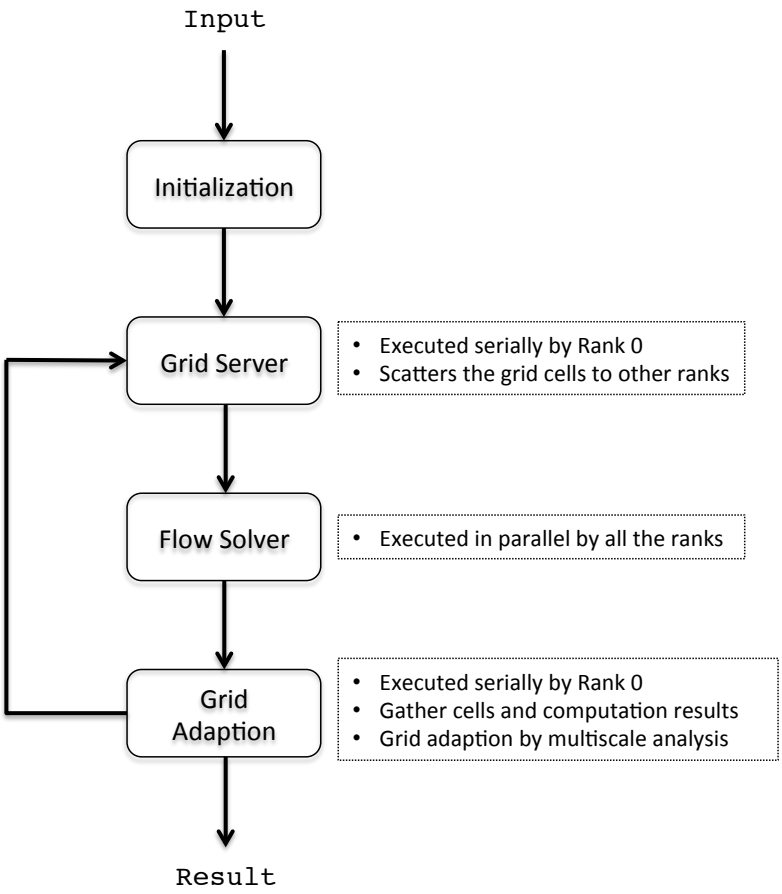


Figure 4.1: The workflow and phases of Quadflow.

Quadflow is an MPI application and its workflow is represented in Figure 4.7. At program start, the MPI processes go through an initialization phase based on the input. Then, rank 0 generates the grid and scatters it equally among the MPI processes through `MPI_Scatter()`. The flow solver is executed in parallel by all the processes. After the computation, the different parts of the grid are gathered at rank 0 through `MPI_Gather()`. A grid adaptation phase is then triggered, which may increase or decrease the number of grid cells depending upon the input and the intermediary results. This is then scattered again to the processes for computation until convergence. The evolving property was realized through the MPI-2 dynamic process management facilities [52]. When additional nodes are allocated, Quadflow spawns new processes on the added nodes using the `MPI_Comm_spawn()` call. The original processes and the newly spawned processes are connected through an inter-communicator. They all collectively participate in an `MPI_Intercomm_merge()` to form an intra-communicator. The spawned processes go through the initialization phase and wait for the cells from rank 0 for computation. Rank 0 now distributes the cells across all the processes (including the newly spawned processes) and the job continues execution.

Although there is an increased programming effort to realize the evolving property in MPI, the program structure of Quadflow facilitates easy data handling and distribution when additional nodes are added. Adaptive programming models like Charm++ or OmpSS usually require negligible programming effort when writing evolving applications. However, the incurred programming effort is not substantial compared to the gain achieved. Applications that

are inherently evolving but unrealized programmatically have to suffer abrupt terminations or incomplete execution within the requested walltime. The user has to resubmit such jobs with a higher allocation. Thus, supporting evolving jobs benefits not only the user but also system providers as re-executions can be avoided, which increases system availability.

4.1.2 Classification and properties of evolving applications

Klein et. al. [73] classified evolving jobs based on the knowledge of a job's evolution as described below.

1. **Fully predictably evolving job**
   A fully predictably evolving job (or simply, predictably evolving job) is one in which the evolving characteristics of a job are well-known in advance. In other words, a user knows when and how many additional nodes will be required by the job. For example, analysis phases that require running a secondary simulation alongside the main simulation can be predicted in advance.

2. **Partially predictably evolving job**
   In a partially predictably evolving job, one can predict either when additional nodes will be required or how many additional nodes will be required. However, it is not possible to predict both.

3. **Unpredictably evolving job**
   As the name indicates, both when and how many additional nodes are required cannot be predicted for an unpredictably evolving job. The evolving behavior may strongly depend on the input.

In general, an unpredictably evolving job can evolve in two ways:

- **Strict evolution** - The application cannot continue execution without the additional nodes. For example, when the memory limit is reached on a node, the application needs additional nodes to offload extra data before being able to continue execution.

- **Non-strict evolution** - The application can continue execution without additional nodes. This typically occurs when more computations are produced due to intermediary results. While additional resources may enable the job to finish before the walltime, the absence of additional resources may require an extension of the allotted walltime to let the application complete.

In both cases, the evolving application can release unused nodes at any point of time. The evolution and predictability depends on the problem being computed. The approaches for scheduling these jobs and the strategy used in this work are discussed in the forthcoming sections.

4.2 Approaches for Supporting Evolving Jobs

In current practice, users execute all the three types of evolving jobs in the following two ways:

- **With statically allocated extra resources**: so that if the job evolves, there are enough resources available to execute the additional computations produced.
- **With statically allocated extra walltime**: so that if the job evolves, there is enough walltime available to execute the additional computations on the same resources.

Both solutions are imperfect and disadvantageous for the system. Although they may avoid incomplete termination of a fully predictably evolving job, any amount of extra resources and walltime may be insufficient for partially predictably and unpredictably evolving job. Furthermore, users must pay for statically allocated extra nodes even for the time they are unused, which leads to substantial resource wastage. Executing jobs with extra walltime is a largely a *guessing-game*. When users anticipating job evolution provide a longer walltime, they may have to endure a longer waiting time under schedulers that assign low priority for long running jobs.

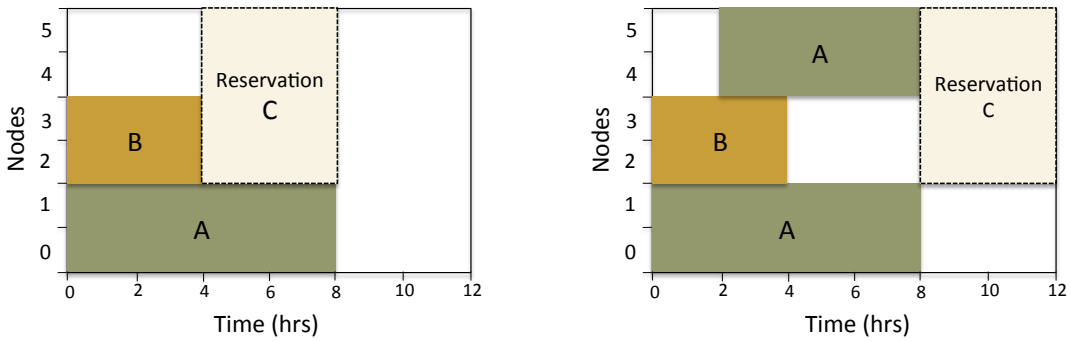
Therefore, dynamic job scheduling and resource management is the most appropriate feature to efficiently schedule evolving jobs. The various approaches for scheduling the different types of evolving jobs and the focus of this work are elaborated in the following sections.

4.2.1 Scheduling fully predictably and partially predictably evolving jobs

The key property that defines a fully predictably or partially predictably evolving job is the *certainty* of the evolution of the job. In other words, additional resources will definitely be used by the job sometime during execution. When the number of additional nodes required can be predicted, they can be preallocated for the job before execution and used at the time of evolution. To improve resource utilization, the preallocated idle nodes can be used to run malleable or preemptive jobs. This guarantees that the evolving job is provided with its preallocated nodes immediately once they are needed. The main disadvantage of this approach is that during the absence of malleable or preemptive jobs, the preallocated nodes will remain unused. However, since it is certain the the job will evolve, it becomes important for the batch system to have sufficient resources available when the evolving job requests them. Therefore, preallocation is the only way to guarantee the resources.

4.2.2 Scheduling unpredictably evolving jobs

Unlike fully predictably or partially predictably evolving jobs, unpredictably evolving job *may* or *may not* evolve, eliminating the certainty aspect of the evolution. Its evolutionary characteristic cannot be predicted. Therefore, dynamic resource allocation requests from such jobs come unexpected and introduce many challenges. In general, a dynamic resource allocation request could be served in the following ways:



(a) State before job A's evolution

(b) State after job A's evolution

Figure 4.2: The effect of the dynamic allocation of nodes to job A on the static reservation of job C.

- Allocating the idle resources
- Allocating resources from a separate partition maintained specifically to serve dynamic requests
- Stealing resources from malleable jobs
- Stealing resources from preemptive jobs

However, even by exercising all the four methods, a dynamic resource allocation request cannot always be satisfied. Since the primary goal of batch job schedulers is to increase throughput and resource utilization, they aim to accommodate as many jobs and maintain the highest system utilization as possible. Therefore, there may not be enough idle resources in the cluster to serve the dynamic request. The separate partition could also be in use completely by other evolving jobs. Furthermore, resource stealing is not feasible when there are no malleable or preemptive jobs at the time of the dynamic request.

Allocating the idle resources to unexpected requests also raises another issue. In practice, production clusters run with a well mixed workload of small and large jobs which often leaves cluster utilization incomplete. This gives an opportunity to allocate idle resources to many dynamic requests. However, it may cause unfair delays to high priority reservations of queued jobs, extending their waiting time. As illustrated in Figure 4.2, consider a cluster system with six nodes in which job A is executing on nodes 0 and 1 for a time slice of 8 hours. Job B acquires nodes 2 and 3, and is scheduled for to run for 4 hours. Queued job C requires 4 nodes and the earliest time it can start is after 4 hours when job B has terminated. Then it could run on nodes 2 to 5. However, if A dynamically acquires the idle nodes 4 and 5 before B terminates, job C will be delayed by an additional 4 hours. Hence, allocating idle resources to dynamic requests can improve system utilization but possibly at the expense of fairness between evolving and rigid jobs. At the same time, improving the availability through the stated methods still cannot guarantee resources to uninformed dynamic requests without prior knowledge of their evolution.

As already stated, the only way to guarantee resources for dynamic requests is through preallocation. A rough estimate of extra nodes can be preallocated to an unpredictably evolving job (e.g., by specifying a minimum and a maximum). To ensure their availability, only malleable or preemptive jobs could be assigned to these resources so that they can be withdrawn when required by the evolving job. However, predicting even the approximate size of the preallocation is error prone. The maximum number of nodes specified for preallocation may be insufficient or result in too few preallocated nodes being used during the job run (with the unused nodes still charged to the user's account). Thus, blocking considerable amount of resources for unpredictably evolving jobs can waste resources and cause queued jobs to starve.

Hence, designing a dynamic scheduler with the primary goal of guaranteeing resources to all the dynamic requests cannot provide good system utilization and can cause higher costs to the users (*guaranteeing approach*). At the same time, designing a dynamic scheduler with the primary goal of improving system performance cannot guarantee resources to all dynamic requests and a dynamic allocation may result in unfair resource usage scenarios, as illustrated in Figure 4.2 (*non-guaranteeing approach*).

4.2.3 Focus of this contribution

As already stated, the performance of a cluster system depends both on the scheduler and the workload. However, the guaranteeing approach depends to a great extent on the workload to achieve good system utilization. Therefore, the disadvantages of the guaranteeing approach are far greater compared to the non-guaranteeing approach in the absence of malleable and preemptive jobs.

Considering the hard-to-predict and non-strict evolving nature of applications like Quadflow, the contribution in this thesis is based on the non-guaranteeing approach where no preallocation is used. If a dynamic request cannot be satisfied with resources immediately, it can either be rejected or answered with an allocation of resources at a later point of time. Jobs sending dynamic requests can be allocated with available idle resources or by stealing resources from malleable jobs. The latter is presented along with the contribution for malleable jobs in Chapter 5. The strategy is complemented by a dynamic fairshare mechanism to ensure fairness between dynamic and static requests through administrator-configurable parameters. Thus, this strategy can be used to accomplish the desired balance among support for evolving jobs, system performance and fair access to resources.

4.3 Dynamic Resource Management for Evolving Jobs with TORQUE

Given the structure of the TORQUE/Maui batch system, the following requirements for supporting dynamic (de)allocation for evolving jobs are imperative.

- An interface through which applications can request/release resources at runtime
- Functionality to queue dynamic requests for scheduling at the TORQUE server
- Functionality to associate/disassociate nodes and jobs dynamically during job runtime

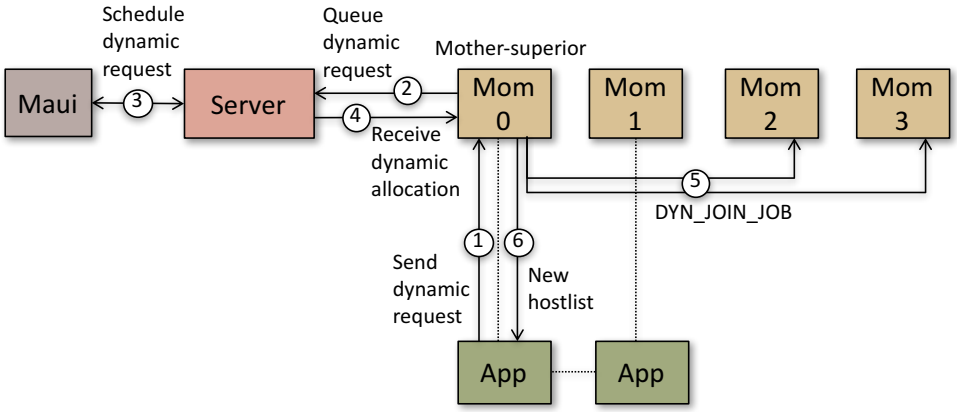


Figure 4.3: Dynamic allocation of nodes 2 and 3. Circled numbers indicate the sequence of steps.

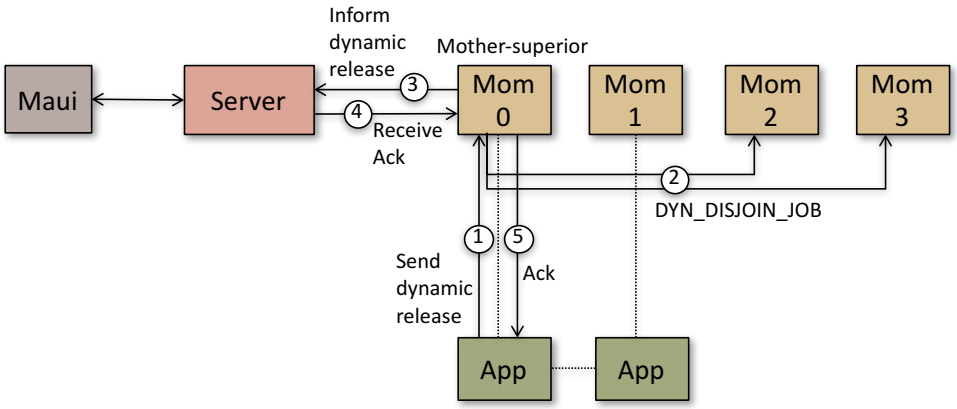


Figure 4.4: Dynamic deallocation of unused nodes 2 and 3. Circled numbers indicate the sequence of steps.

We extended TORQUE by adding the above features and the workflows of dynamic allocation and deallocation is illustrated in Figures 4.3 and 4.4 respectively. For a dynamic allocation, applications can use the `tm_dynget()` function of the extended TM interface by specifying the number of nodes and processors per node. The mother-superior forwards the request to the `server` which changes the job to a special *dynqueued* state. This triggers a new scheduling cycle and additional resources are allocated for this request. The `server` forwards the new hostlist, which is a list of new nodes added to the job, to the mother-superior and changes the job state back to *running*. The hosts from the existing allocation and the dynamically allocated hosts perform a *dyn_join* operation which expands the resource allocation for the job. The mother-superior then responds to `tm_dynget()` with the dynamically allocated hostlist. MPI applications can use the MPI-2 dynamic process management facilities to spawn new processes on the additionally allocated nodes. MPI implementations offer a "host" or "add-host" parameter to the MPI_Info argument to specify a newly allocated hostlist. Similarly, the function `tm_dynfree()` can be called to release nodes by passing the list of nodes to be released as a parameter (illustrated in Figure 4.4).

The call `tm_dynfree()` usually returns true, as a release operation is rarely unsuccessful. During dynamic deallocation, the `moms` perform a *dyn_disjoin* operation with the nodes to be

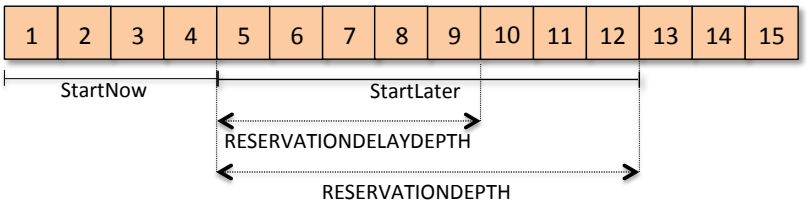


Figure 4.5: The number of *StartNow* and *StartLater* jobs in a queue. In this example, `ReservationDepth` is longer than `ReservationDelayDepth`.

released and the `server` is informed of the deallocation. Finally, the `server` updates the freed node's states internally, after which they can be allocated to other jobs.

Basically, any process from any host of the parallel job can call `tm_dynget()` to request new resources through its local `mom`. However, to ensure that only one dynamic request from the same job is pending at the `server` at a time, the dynamic requests are always forwarded to the `server` through the mother-superior. This simple API consisting of two functions is sufficient for dynamic resource (de)allocation.

4.4 Scheduling Evolving Jobs with Maui

By design, the Maui scheduler supports scheduling of rigid jobs only. In our work, the Maui scheduler was extended to schedule dynamic requests by:

- Enriching Maui's iteration with a scheduling algorithm that also supports dynamic requests
- Enhancing the resource allocation mechanism to allocate resources for dynamic requests
- Implementing a dynamic fairness scheme to ensure fairness between dynamic and static requests

The extended Maui iteration is detailed in Algorithm 2. In each iteration, Maui obtains the workload and resource information from TORQUE and prioritizes a list of eligible static jobs and dynamic requests separately (steps 2-9). While the static jobs are prioritized according to normal priority factors, the dynamic requests are prioritized in FIFO order. The static jobs are then scheduled and the necessary reservations are created but the jobs are not started immediately (step 10). The reserved static jobs can be classified into two categories: (i) *StartNow*: jobs that can be started immediately, and (ii) *StartLater*: jobs that can be started only at a later point of time. In the next step, for each dynamic request in the queue, the scheduler tries to allocate the idle resources and measures the delays that may be caused to the *StartNow* and *StartLater* jobs.

In the original algorithm, the number of *StartLater* jobs produced is determined by the `ReservationDepth` parameter in order to compute the backfill windows. The extended algorithm provides a `ReservationDelayDepth`, which specifies the number of jobs for which delays need to be computed and considered when scheduling dynamic requests. Therefore, the number of *StartLater* jobs in the extended algorithm is determined by the maximum of `ReservationDepth` and `ReservationDelayDepth`. This is illustrated in

Algorithm 2 Extended Maui Iteration

```
1:  while TRUE do
2:      Obtain resource information from TORQUE
3:      Obtain workload information from TORQUE
4:      Update statistics
5:      Refresh reservations
6:      Select static jobs eligible for priority scheduling
7:      Select dynamic requests eligible for priority scheduling
8:      Prioritize eligible static jobs
9:      Prioritize eligible dynamic requests
10:     Schedule static jobs in priority order and create reservations (without job start)
11:     for each dynamic request in the queue do
12:         Try to allocate resources for dynamic request (from idle before preemptible nodes)
13:         if resources are available for the job then
14:             Check dynamic fairness policies to determine if job is allowed to get resources
15:             if job is allowed then
16:                 Continue dynamic job with expanded resource allocation
17:                 Update dynamic fairshare statistics
18:             else
19:                 Reject the dynamic request
20:             end if
21:         else
22:             Reject the dynamic request
23:         end if
24:     end for
25:     Schedule the static jobs in priority order and create reservations (with job start)
26:     Backfill static jobs
27: end while
```

Figure 4.5. Thus, the algorithm uses `ReservationDepth` number of jobs in the list of *StartLater* jobs when computing backfill windows and `ReservationDelayDepth` number of jobs when computing delays caused due to dynamic requests. This allows delays to be computed for a controlled number of jobs irrespective of whether a conservative backfilling with large `ReservationDepth` is deployed or optimistic backfilling with lower `ReservationDepth` is deployed. Similar to `ReservationDepth`, a proper choice of `ReservationDelayDepth` for a site depends on its workload characteristics.

For each dynamic request, if the dynamic request can be satisfied, the dynamic fairness policies are invoked to determine whether the allocation is fair and can be allowed (steps 11-14). The dynamic fairness parameters are site-configurable parameters and are described in Section 4.4.1. If the reservation is allowed, the dynamic fairness statistics are updated and the set of nodes allocated to the job is expanded (steps 16-17). If not, the dynamic request is rejected (step 18). When all dynamic requests have either been satisfied or rejected, the static jobs are scheduled and started in the priority order (step 25). In this step, the number of jobs

started may be different than the number of *StartNow* jobs in the previous step due to resources allocated to dynamic requests. Thereafter, low priority jobs are backfilled out-of-order (step 26).

Strategies for allocating resources in response to dynamic requests can be controlled by site-specific parameters. For example, a dynamic request may obtain resources by preempting (when enabled) other jobs, for example, jobs with low priority or backfilled jobs. Existing Maui parameters can be used for this purpose. In the current version, due to the simple dynamic (de)allocation protocol, applications that cannot continue without an expanded set of resources must request for resources again at a later point in time if rejected. In contrast, leaving the dynamic request queued at the `server` and blocking the application until resources are obtained is not the best choice for evolving jobs that can continue execution but would have to run longer without more resources. An efficient negotiation mechanism where the application can specify a timeout for obtaining resources and where the batch system can indicate the time of availability of resources would be beneficial, and is one of our future goals.

4.4.1 Dynamic fairness policies

Fair sharing of resources between users is a compulsory responsibility of a site and is realized through job, user, and resource accounting. In static scheduling, fairness policies play a decisive role in the prioritization of jobs at most sites, as supercomputing resources are shared by an extensive group of users. The Maui scheduler's fairshare policies, configurable through a set of administrator parameters, allow fine-tuned control of resource sharing among different users, groups, accounts, classes and qualities of service [65]. However, when adding support for evolving jobs, these parameters cannot be used to control the ill effects of resource stealing by an unpredictably evolving job.

For the dynamic scenario, we introduce two types of fairness policies dictated by the new parameter `DFSPolicy`: (i) `DFSSingleJobDelay` and (ii) `DFSTargetDelay`. The `DFSSingleJobDelay` simply imposes a limit on how long each queued job of a particular user can be delayed by dynamic allocations to evolving jobs. The limit can be different for every user and can be set by the `DFSSingleDelayTime` parameter.

On the other hand, the `DFSTargetDelay` policy limits the cumulative delay caused to users over a configurable interval. The delay is set with the `DFSTargetDelayTime` parameter and the interval with the `DFSInterval` parameter, both in total seconds or `HH:MM:SS` format. The dynamic fairness setting can also be configured to combine both policies or disabled by setting the `DFSPolicy` to `DFSSingleTargetDelay` or `NONE`, respectively. When disabled, the dynamic requests will have the highest priority over the static jobs and the delay caused to static jobs will be ignored. Furthermore, the `DFSDynDelayPerm` parameter (1: allow, default ; 0: disallow) specifies whether a particular user's job can be delayed or not due to dynamic requests. Thus, a dynamic allocation will be unsuccessful if it would delay a job that is not authorized to be delayed. Also, when the evolving job and the static job are from the same user, the delay is not considered.

After each interval, the current delays are rolled back according to the `DFSDecay` parameter. This parameter indicates how much the current delay should decay at the end of an interval.

```
DFSPOLICY             DFSSINGLEANDTARGETDELAY
DFSINTERVAL           06:00:00
DFSDECAY              0.4

USERCFG[user01]       DFSDYNDELAYPERM=1 DFSTARGETDELAYTIME=3600 \
                      DFSSINGLEDELAYTIME=0
USERCFG[user02]       DFSDYNDELAYPERM=0
USERCFG[user03]       DFSDYNDELAYPERM=1  DFSTARGETDELAYTIME=0 \
                      DFSSINGLEDELAYTIME=00:30:00
USERCFG[user04]       DFSDYNDELAYPERM=1 DFSTARGETDELAYTIME=02:00:00 \
                      DFSSINGLEDELAYTIME=00:15:00
GROUPCFG[group05]     DFSTARGETDELAYTIME=04:00:00
GROUPCFG[group06]     DFSDYNDELAYPERM=0
```

Figure 4.6: An example of dynamic fairness configuration.

For example, consider the limit of delay for a user to be 4800 seconds for an interval and the current delay at the end of the interval to be 3600 seconds. If the `DFSDecay` is set to 0.2, then the current delay in the next interval will be initialized by 20% of 3600 seconds, which is 720 seconds.. Therefore, the user's jobs can be delayed for a maximum of 4080 seconds in the new interval. This parameter allows historical delays to be considered.

Figure 4.6 shows an example configuration of the `DFSSingleAndTargetDelay` policy over an interval of 6 hours with a decay of 0.4. Basically, the above delay permission and time settings can be set not only for users but also for groups, accounts, job classes and qualities of service of the jobs. In an interval, assuming the current delay to be 0 for all users and groups, `user01`'s jobs can be delayed for any amount of time but cumulatively `user01` may experience only a maximum of an hour's delay. On the other hand, `user03` has no limit on the cumulative delay but each of `user03`'s jobs can only be delayed by a maximum of half an hour. `User04`'s limits combines both methods where the user can only experience up to 2 hours of cumulative delay but each job may only be delayed by 15 minutes at most. The `group05`'s configuration limits the cumulative delay experienced by all the users belonging to the group to a maximum of 4 hours. When limits are specified for both an individual and his group, the most restrictive limit is used. Finally, jobs of `user02` and users of `group06` are not allowed to be delayed by dynamic allocations. These simple parameters easily enable the desired dynamic configuration for a site according to its job mix. The parameters can be used to effectively avoid the starvation of static jobs.

An aspect to be taken into consideration for the careful choice of delay limits is the effect of the difference in walltime and actual execution time of evolving jobs. Users choose walltimes that are usually greater than the actual execution time of the application. Dynamically requested nodes are by default allocated for the rest of the walltime limit of the running evolving job. Since the actual execution time might be less than the walltime originally specified by the job, the delays computed for the queued jobs when satisfying an dynamic request may also be larger than the actual delay that they might experience. Furthermore, the evolving application could finish even faster with additional dynamic resources, causing even larger gap between the delay computed and the actual delay experienced by the queued job. Therefore, the delay limits should be configured with moderately higher values than intended to handle such instances. This would enable a more accurate fairness measure. Also, when if a user attempts

to take advantage of the system by submitting a small job in order to higher priority and keep expanding after job start, the user's next job will receive a low priority.

4.5 Evaluation

In this section, we evaluate the proposed batch system. We use Quadflow to prove our concept and show the benefits that dynamic allocation can deliver for certain groups of production applications. We further present an analysis of a dynamic workload from both the user and the system perspective, using the ESP benchmark suite modified to contain evolving jobs.

The evaluation consists of real experiments and is not based on simulations. All the experiments were conducted on a 15-node cluster system equipped with 2 Intel Xeon X5570 processors per node running at 2.93 GHz (8 cores per node). A separate 16th node was used as the headnode running the modified TORQUE version 4.1.0 and Maui version 3.3.1. The same node was also used as the frontend. As MPI implementation, we used Open MPI version 1.7.3.

4.5.1 Quadflow

As described earlier, the MPI-based CFD flow solver Quadflow solves the compressible Navier-Stokes equations using a cell-centered fully adaptive finite volume method on locally refined grids. Starting on the coarsest grid level, the computational grid of the investigated flow configuration is successively refined until the final grid level is reached. The local refinement of the grid leads to high computational efficiency. However, since the areas in need of refinement can only be identified during the solution process, no prior knowledge is available on the development of the number of grid cells. Two generic test cases are investigated in the following: (i) FlatPlate: the laminar boundary layer flow over a flat plate in a supersonic flow field at Mach 2.6 is a pertinent example of a generic validation test case [109]. The boundary layer requires a high local resolution, whereas large parts of the flow domain can be kept quite coarse. (ii) Cylinder: the supersonic flow around a 2D Cylinder at Mach 5.28 is a typical example of a high-enthalpy stagnation point problem. Such flow fields are characterized by strong bow shocks, which need to be captured again with high local resolution. However, the exact location and size of these shocks is not known apriori which makes it difficult to predict the required number of grid cells in advance. Realistic scenarios frequently involve shock-shock interactions [110], in which the aforementioned problems become even more severe.

Figure 4.7 shows the execution times of the two cases in a static scenario with 16 and 32 cores (8 processes per node), and a dynamic scenario where the execution is started with 16 cores/processes and expanded to 32 cores/processes at a threshold point. After each grid adaptation, the next computation phase is shaded lighter than the previous one. Technically, both cases use different numerical methods and the computational intensity of the FlatPlate case with one cell is equivalent to the Cylinder case with 4-5 cells. In the dynamic scenario, the dynamic allocation was done when a grid adaptation step led to more than 3000 cells per process for the FlatPlate and 15000 cells per process for the Cylinder test case. The application performed a total of 2 and 5 adaptations for the FlatPlate and the Cylinder test case, respectively. The

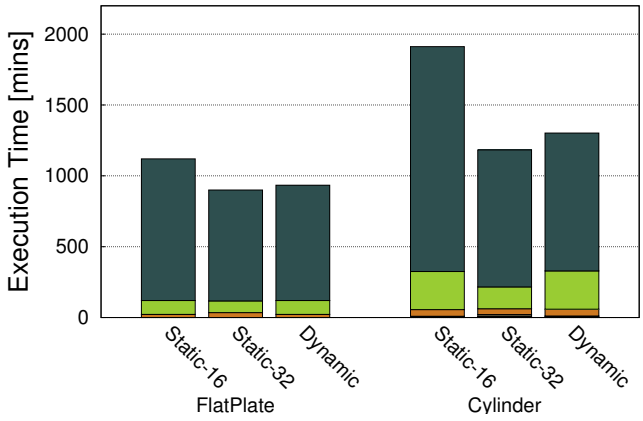


Figure 4.7: Execution times of static and dynamic Quadflow test cases broken down by adaptation phase. Same shades denote the same phase.

threshold for the number of cells per process was exceeded in the final grid adaptation phase in both cases. That is, a dynamic request was issued after the last grid adaptation.

We can observe that by expanding its allocation to twice the number of allocated cores, the Cylinder test was faster by 33% (saving 10 hours) and the FlatPlate by 17% (saving 3 hours). The applications could also have been started with a larger allocation of 32 cores to obtain the speedup displayed without any dynamic allocation. However, this is only possible if a user can predict the threshold-exceeding growth of cells per process. A larger static allocation may also lead to under-loaded resources with too few cells per process as can be seen in our example. For instance, for the FlatPlate case, we can see that the time taken until the final grid adaptation level is identical when executed with 16 or 32 cores. This implies that starting the execution with 32 cores (i.e., with an extra 16 cores) has no effect as long as the number of cells stay within the threshold. By using resources only when required, such applications can obtain a similar speedup compared to starting the execution with a larger allocation. This not only reduces the usage costs for the user but also allows unused resources to be allocated to other jobs, thereby improving system utilization and throughput. These aspects are studied in the next section.

4.5.2 Dynamic ESP benchmarks

A meaningful inference of scheduling performance can be obtained by only analyzing the scheduling outcome of a given workload. Given the scope of this work, a workload consisting of rigid and evolving jobs is necessary to evaluate the proposed batch system. Common scheduler evaluation benchmark workloads contain only rigid jobs. We are not aware of any benchmark with evolving jobs that is capable of assessing dynamic scheduling quality. Therefore, we modified the well-known ESP benchmark [46] so that it consists of both evolving and rigid jobs for our workload. Considering applications like Quadflow, we mainly focus on dynamic allocation of nodes rather than their dynamic deallocation.

The original ESP benchmark is composed of 230 jobs with 14 different job types each running the same synthetic application. Each job type has a unique fixed execution time and uses a

Table 4.1: The various job types of the modified ESP benchmark, their resource requirements, their static execution time (SET) and dynamic execution time (DET).

Job type	User	Size	Count	Evolving Job	Static Execution Time [secs]	Dynamic Execution Time [secs]
A	user01	0.03125	No	75	267	-
B	user02	0.06250	No	9	322	-
C	user03	0.50000	No	3	534	-
D	user04	0.25000	No	3	616	-
E	user05	0.50000	No	3	315	-
F	user06	0.06250	Yes	9	1846	1230
G	user06	0.12500	Yes	6	1334	1067
H	user06	0.15820	Yes	6	1067	896
I	user06	0.03125	Yes	24	1432	716
J	user06	0.06250	Yes	24	725	483
K	user07	0.09570	No	15	487	-
L	user08	0.12500	No	36	366	-
M	user09	0.25000	No	15	187	-
Z	user10	1.00000	No	2	100	-

fraction of the total resources. The benchmark was modified to contain 30% evolving jobs and 70% rigid jobs (totaling to 69 evolving and 161 rigid jobs). Each rigid job type was considered to be run by a unique user and all the evolving jobs were considered to be executed by the same user as listed in Table 4.1. Job types F, G, H, I and J are considered as evolving jobs and the time at which the dynamic request is sent is modeled as in the Cylinder case of Quadflow. From the complete static and dynamic run of the Cylinder test case, it can be derived that a dynamic allocation is needed after 16% of the total static execution time. Therefore, F, G, H, I and J jobs request 4 additional cores each after 16% of their total static execution time according to the ESP benchmark. When resources are not available at that point, the job continues and requests resources again after 25% of the total static run time as a second chance to obtain resources. If both attempts fail, the job continues with the current allocation. If the dynamic allocation is successful, a linear reduction of the execution time for the evolving job is assumed similar to the Cylinder case of Quadflow. Jobs are submitted in a particular order with the first 50 jobs submitted instantly. Thereafter, jobs are submitted one by one with an interval of 30 seconds between each job submission. The workload consists of 2 special Z type jobs which use the complete cluster. After submitting the other 228 jobs, the Z jobs are submitted 30 minutes after the last job submission. As defined by the ESP benchmark, once the Z jobs are submitted, they receive the highest priority in the queue and no other low priority job can be executed. Backfilling is also disabled for the period that a Z job is queued. Evolving jobs that are already running may still obtain resources dynamically during this phase. The corresponding static execution time (SET) and the dynamic execution time (DET) are also listed in Table 4.1.

Table 4.2: Performance comparison of the evaluation configurations.

Config	Time [mins]	Satisfied Dyn Jobs	Util [%]	Throughput [Jobs/min]	Throughput [% Increase]
Static	265.78	0	77.45	0.86	-
Dyn-HP	238.78	43	85.02	0.96	11.3
Dyn-500	248.85	20	82.26	0.92	6.8
Dyn-600	241.06	27	83.57	0.95	10.2

Four configurations were used for our evaluations. First, a static workload where F, G, H, I and J do not acquire any dynamic resources. Second, a dynamic workload with dynamic fairness disabled, thus giving dynamic requests highest priority (Dyn-HP). In the third configuration, a dynamic fairness policy limited the cumulative delay for each user's static job to 500 seconds in an interval of 1 hour (Dynamic-500). Similarly, the fourth configuration limited the cumulative delay for each user's static jobs to 600 seconds (Dynamic-600). In all the configurations, the `ReservationDepth` and `ReservationDelayDepth` parameters were set to 5. Table 4.2 lists the various performance characteristics of the four workload configurations.

The first two columns of Table 4.2 show the total execution time of the workload (in minutes) and the number of evolving jobs that succeeded with their dynamic requests. The highest priority configuration (Dyn-HP) achieves the best overall system performance. 43 out of 69 evolving jobs obtained dynamic resources and the workload execution time was 10% faster (27 minutes). The system utilization increased to 85% as compared to 77% in the static setting and the throughput (TP) increased by 11.3%. Although the configuration improves the overall performance, it does not consider the delays incurred by static jobs.

Figure 4.8 shows the effects of such a configuration. It compares the waiting time of jobs (in the order of job submission) in the static workload with the dynamic workload in the Dyn-HP configuration. It is evident that due to better resource utilization and earlier completion of evolving jobs, the overall waiting time of several jobs is reduced. However, we can see that many jobs with job IDs between 70 and 125 experience longer waiting times as compared to the static scenario. This unfairly affects the users who submitted jobs in this range. This can be observed in Figure 4.9, which compares the waiting time of type L jobs in the order of their submission. Half of the type L jobs are affected by longer waiting times. For other large production workloads consisting of long running jobs, these delays are more severe for certain users. Thus, obtaining the highest performance leads to such undesirable consequences.

The dynamic fairness policies address such issues. Figure 4.10 compares the waiting times in the Static, Dyn-HP and Dynamic-500 configurations. The waiting time of jobs can be observed to be more uniform with respect to the static scenario. Figure 4.9 also shows the considerable improvement that type L jobs experienced due to this strategy. However, the configuration satisfied only 20 of the 69 evolving jobs, which reduced throughput and system utilization (Table 4.2) compared to the highest priority configuration. This is a natural consequence of enabling a weak fairness scheme.

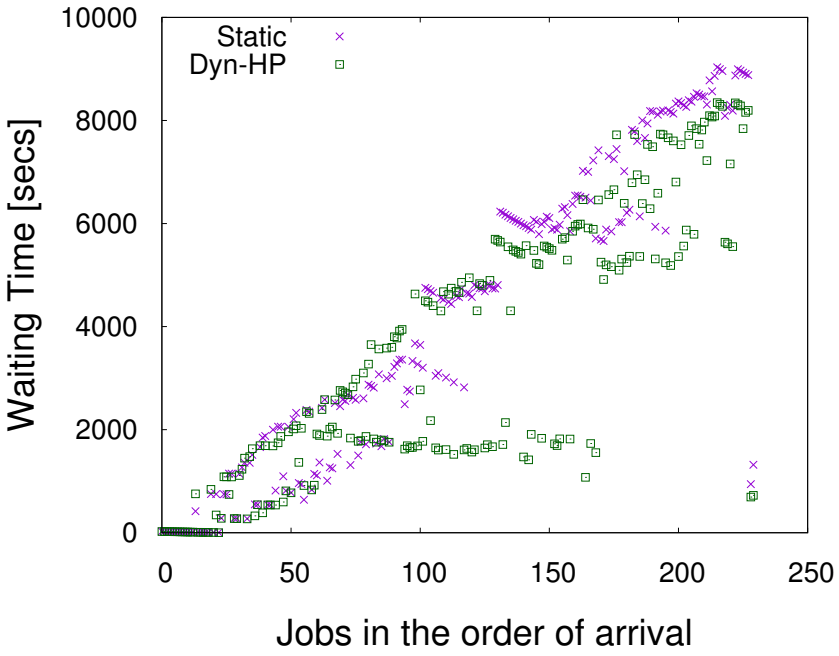


Figure 4.8: Comparison of the waiting times of jobs in the static and dynamic workload where highest priority is used for dynamic requests.

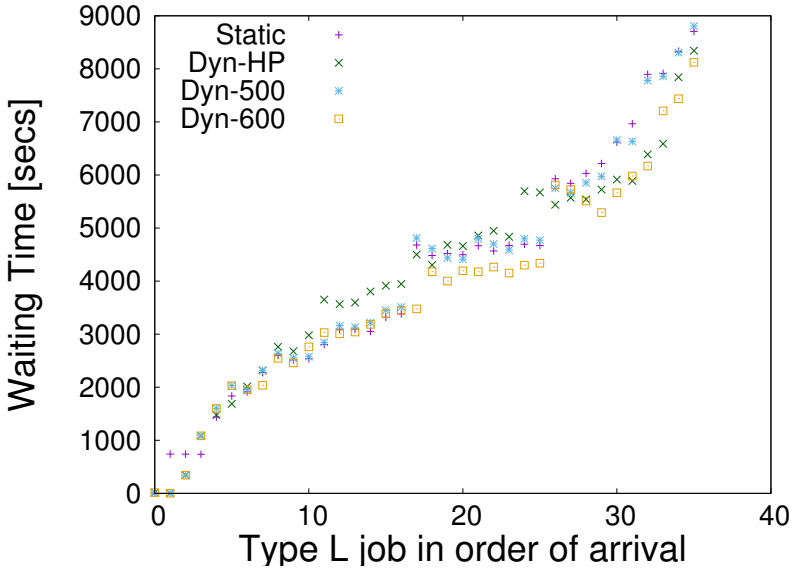


Figure 4.9: Comparison of waiting times of type L jobs in all four configurations.

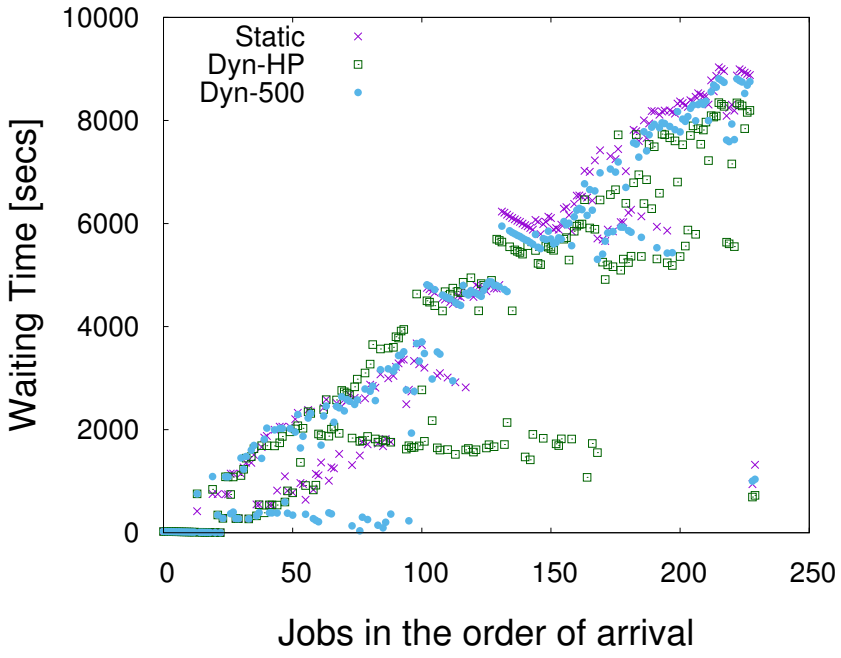


Figure 4.10: Comparison of waiting times of jobs in configurations Static, Dyn-HP and Dyn-500 .

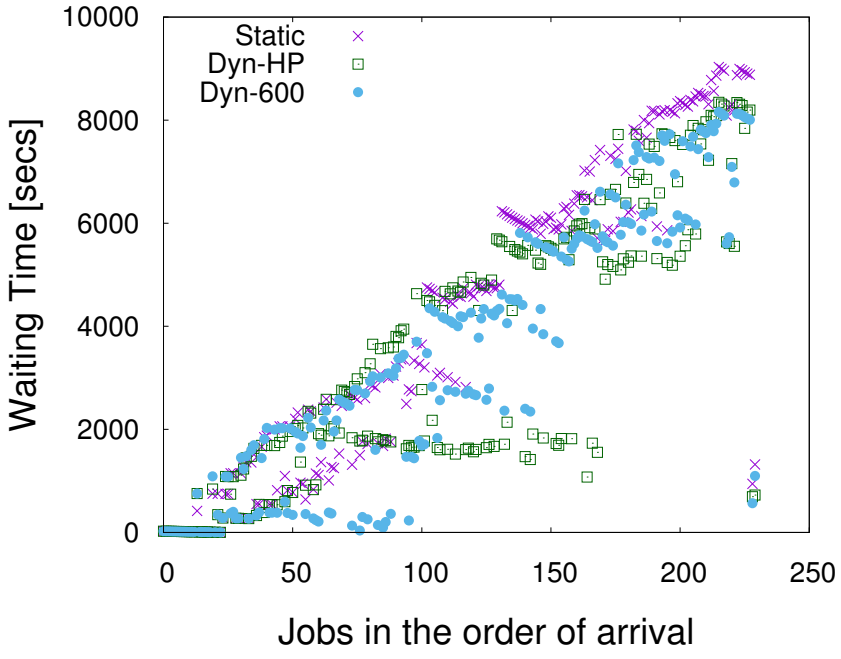


Figure 4.11: Comparison of waiting times of jobs in configurations Static, Dyn-HP and Dyn-600.

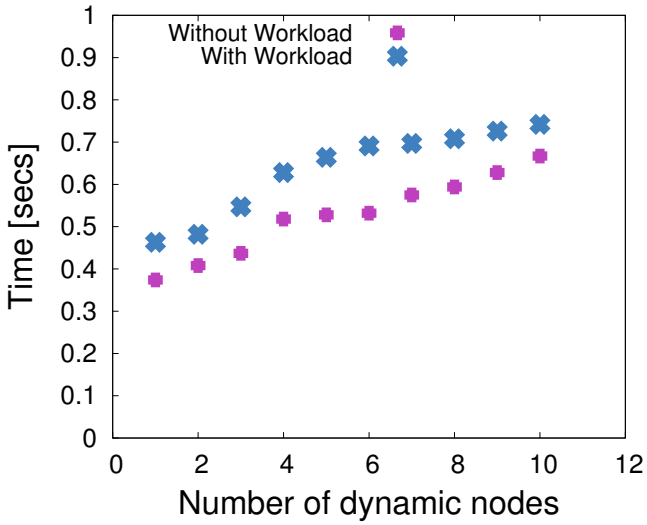


Figure 4.12: Time taken for the dynamic allocation of 1 to 10 nodes from a job using one statically allocated node.

However, moderate fairness policies can better balance system performance and user fairness. Figure 4.11 compares the waiting time of the Static and Dyn-HP with the Dynamic-600 configuration. We can observe that with a little less restriction the number of successful dynamic requests increased to 27 and a system utilization and throughput close to that of the Dyn-HP configuration is realized (Table 4.2).

An important aspect that leads to this result is also the backfilling strategy. Our dynamic scheduling algorithm prefers to allocate idle resources to dynamic requests over backfilling the resources for smaller low-priority jobs (as long as the dynamic request satisfies the fairness condition). This may give the impression that fewer jobs are backfilled in such dynamic environments. Our results show the opposite. There may be idle resources or resources may become idle shortly after responding to all dynamic requests. These may not be enough to service the high priority job in the queue (delayed due to dynamic requests). However, it allows more smaller jobs to be backfilled, which leads to higher throughput. In Figures 4.8, 4.10, and 4.11, the backfilled jobs are the ones with considerably lower waiting times in the mid range of the job IDs. The Dyn-HP configuration backfills the greatest number of jobs, followed by the Dyn-600 and Dyn-500 configurations. That is, the larger the number of successful dynamic requests, the greater was the backfilling ability and the higher was the throughput (cf. Table 4.2). Nevertheless, this pattern largely depends on the workload and may vary for a workload which can be well packed in the system for high system utilization. Thus, in the scenario of scheduling unpredictably evolving jobs, the results show that our approach provides a robust and flexible way to obtain a good balance between system performance and fairness.

4.5.3 Dynamic allocation overhead

The gain for an evolving application also depends on the overhead of the dynamic scheduling mechanism. Figure 5.7 shows the overhead of allocating from 1 to 10 nodes dynamically from

a job running on one statically allocated node. Two scenarios are compared: (i) dynamic allocation without any workload at the batch system and (ii) with a workload of rigid jobs and a `ReservationDelayDepth` parameter of 5. It can be observed that the overhead for the dynamic allocation of as many as 10 nodes lies only in the sub-second range, which is negligible for a real-world application.

4.6 Summary and Conclusion

Dynamic resource-management facilities are key to serving the needs of the growing complexity of applications. A lack of these facilities may even limit the application domains that can be explored. Therefore, it is indispensable for future systems . Given the different types of evolving jobs and their characteristics, many challenges have to be overcome in order to develop a batch system that can efficiently schedule all the types of evolving jobs together.

In this contribution, we have demonstrated a batch system capable of on-the-fly resource allocation for unpredictably evolving jobs based on runtime requests while ensuring fair access to resources for rigid and evolving jobs. The batch system can expand and shrink resource allocations to jobs with little overhead. Results show that supporting evolving applications can lead to reduced waiting and turnaround times while increasing resource utilization and throughput. Moreover, the dynamic fairness policies provide a simple set of parameters to configure fairness metrics according to site-specific requirements. We believe this contribution takes the research in resource management a step forward towards addressing the demands of current and forthcoming HPC applications, systems and practices.

5 Supporting Malleable Jobs

This chapter describes the dynamic resource management and job scheduling techniques developed for supporting malleable jobs in a cluster environment. It presents an overview of malleability in HPC applications followed by a brief description of the aspects and approaches in scheduling malleable jobs. Thereafter, methods for dynamic resource management and the scheduling algorithm designed for malleable jobs are detailed. After evaluating the proposed approach with a modified ESP benchmark, the chapter concludes with a summary.

5.1 Malleability in HPC Applications

Unlike evolving applications, malleability is harder to realize as it requires the application to adapt to changes triggered from an external entity (i.e., the batch system). In general, an application can become malleable as a result of the following: (i) use of an adaptive programming model, and/or (ii) using adaptive algorithms.

Adaptive programming models usually enable automatic malleability of applications. In other words, the parallel runtime executes the application in a way that it fully supports resource changes. Charm++, OmpSs, Adaptive MPI and OpenMP are prominent examples. The principal goal behind the adaptive nature of the parallel runtime systems is, however, not malleability itself. Malleability comes as a by-product of adaptivity enabled for features such as load balancing, fault tolerance and energy efficiency. Effective and automatic balancing of computations across multiple nodes and cores of a node is still an active area of research. Different applications require distinct ways of load balancing, which parallel programming systems try to establish without user intervention. Moving towards exascale, energy efficiency would benefit from parallel runtimes that can run an application in a flexible manner. Furthermore, as exascale systems are predicted to have high failure rates, it is important that parallel programming systems make the application flexible enough to endure unexpected resource losses.

Adaptive algorithms also support malleability of applications. Typically, the master/slave way of programming enables malleability of the application. However, implementing it under a rigid programming model may disallow malleability. An example is the application Quadflow which uses a master/worker scheme but is not malleable because of the usage of MPI. Making MPI programs malleable requires self-developed or external libraries that can capture expand/shrink messages and then trigger MPI-2 process management facilities to modify the MPI communicators. This demands increased programming and shrink/expand overhead, since new MPI communicators need to be established for every shrink/expand operation. Examples are the WaterGap application [111], which performs global assessment and prognosis of water availability, and an astronomical application [112, 113], which finds stellar objects from the datasets of the Milky Way galaxy. Iterative applications can be made malleable with little effort on the algorithmic front. However, this also requires the programming model to support adaptivity.

Efforts on making malleable iterative MPI applications have also been reported [54, 114]. Applications that use AMR and multiscale analysis techniques can exhibit malleability for certain problems.

Thus, adaptivity at the level of the programming model is an important aspect towards realizing malleability for a larger scope of applications. An adaptive programming model can automatically make an application malleable unless the algorithm used by the application demands a rigid execution. In this chapter, we use Charm++ to investigate malleability and devise resource management and scheduling schemes.

5.2 Approaches for Scheduling Malleable Jobs

In this section, we discuss the various aspects to be considered when designing a malleable job scheduler and highlight the goal behind the approach taken in this work.

5.2.1 Resource utilization and throughput

Malleable jobs help considerably reduce resource wastage as they can potentially use the idle resources effectively when expanded. However, increased resource utilization does not always imply higher throughput. When a cluster is running several malleable jobs, an inefficient choice of job for expansion or shrinkage can lead to higher resource utilization without any increase in throughput. In certain cases, it may even be counterproductive to a gain in throughput. Such scenarios are shown in Section 5.5 with the evaluation of some of the malleable job scheduling strategies. Therefore, a malleable job scheduling scheme must analyze job and resource dependencies to deliver high throughput alongside better resource utilization.

5.2.2 Fairness

Enabling some degree of fairness in expand/shrink operations is essential as it can motivate users to write more malleable applications as opposed to rigid ones. Equipartitioning is a reasonably good strategy towards enabling fair dynamic (de)allocations. However, equipartitioning alone cannot improve the global system throughput and response times for the same reason that it can contradict the selection of best malleable for expand/shrink operations. This is also exemplified with experiments in Section 4.5. Therefore, a good malleable scheduling strategy must target system efficiency along with as much fairness as can be delivered.

5.2.3 Communication with the parallel runtime system

Apart from powerful scheduling schemes, enabling malleability also requires a scalable shrink/-expand mechanism. Typically, expansion can happen almost instantaneously as the parallel runtime may be able to spawn new parallel tasks as soon as it obtains the fresh nodes. However, shrinking may require more time as it involves waiting until the task running on the nodes to be removed to be completed. Thereafter, the data required by rest of the application

from these processes needs to be saved before the processes can be killed. To facilitate immediate release, it is also possible to use the internal checkpointing mechanism of Charm++ to abort the processes immediately and restart the application from the latest checkpoint. In our approach, for simplicity, we do not use checkpoint/restart mechanism and therefore let the scheduler wait for the tasks running on the nodes to be removed to complete during a shrink operation. Another aspect of communicating with the runtime system is the option of making scheduling decisions based on application feedback. Typically, when running iterative applications, sampling the execution time of iterations for different number of processes can help the batch system select more responsive and well-scaling applications for shrink/expand operations. However, feedback mechanisms introduce other overheads such as too frequent communication, inconsistency (as iteration times are not always constant), and increased complexity for non-iterative malleable applications. Efficient feedback mechanisms for malleable applications have been exclusively studied [115, 116]. Scheduling malleable jobs based on feedback from application is out the scope of this work.

5.2.4 Summary of the approach taken in this work

The main goal behind the design of the malleable scheduler in this work is to improve resource utilization and throughput. In this regard, fairness in distributing resources to malleable jobs is only given a second priority. This does not interfere with the regular priority policies set by an administrator for job submissions. Our approach does not consider checkpoint/restart mechanisms in expand/shrink operations. Jobs expanded with additional nodes will be able to use the newly added nodes after the application's next synchronization step. Similarly, when jobs are shrunk, the scheduler waits until the application reaches a synchronization point and can give away the nodes that are to be released.

5.3 Dynamic Resource Management for Malleable Jobs with TORQUE

Given the structure of the TORQUE/Maui batch system, the following features were implemented in TORQUE to enable shrink/expand facilities:

- An extended `qsub` command to submit a malleable job
- Functionality to shrink/expand a resource set at the `server` based on Maui's instruction
- Functionality to associate/disassociate nodes at the `mom` based on the instruction from the `server`
- A communication mechanism between the `mom` and the Charm++ runtime system for malleability interaction

A malleable job can be submitted with the extended `qsub` command as shown in the example below:

```
$ qsub -l nodes=2:ppn=8,walltime=3600 \
> -L max=6,type=charm++ jobscript.sh
```

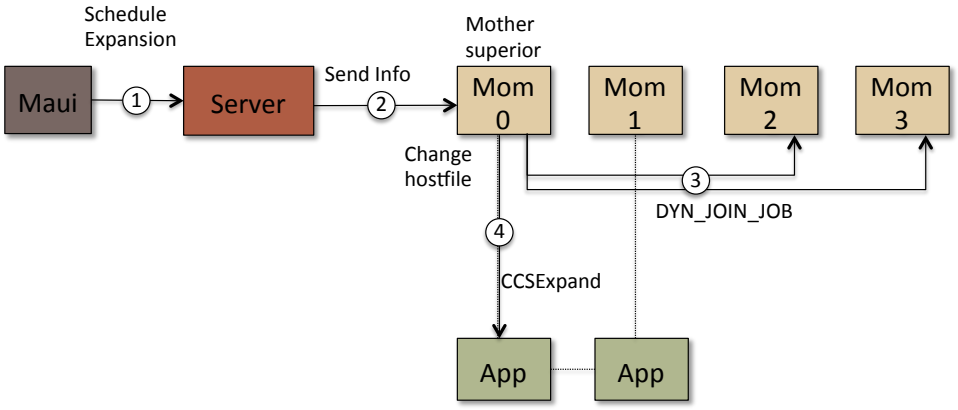


Figure 5.1: Expanding a job by adding nodes 2 and 3. Circled numbers indicate the sequence of steps.

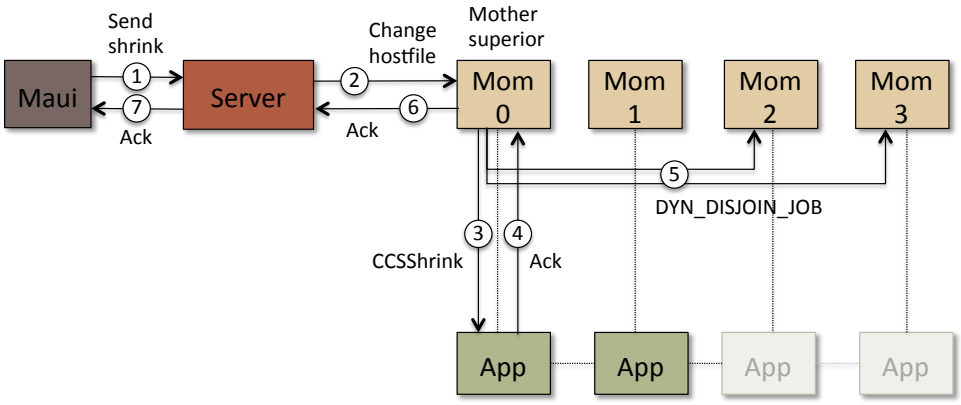


Figure 5.2: Shrinking a job by removing nodes 2 and 3. Circled numbers indicate the sequence of steps.

A user indicates the minimum number of nodes required for a job, the fixed number of processors required per node and the duration of the job with the minimum number of nodes through the `-l` option. To denote the malleability of the job, the user must specify the `-L` option indicating the maximum number of nodes that can be used by the job and a job *type*. In general the shrink/expand facilities can be used for any job. However, as there is no standard way of interacting between the batch system and the parallel runtime, it requires development and integration of appropriate communication for every programming paradigm. The job *type* hints the type of programming paradigm used by the job to the batch system so that the right mechanism can be chosen for communication. In the current version, only Charm++ jobs are fully supported.

For malleability interactions, the Converse Client-Server interface (CCS) [117] in the Charm++ runtime system was leveraged and a shrink/expand specific handler was developed. A separate management thread of a Charm++ job acts as a CCS-server that listens to shrink/expand commands via TCP/IP as soon as the application begins executing. The corresponding CCS-client has been integrated into the `mom`. Before starting the application through the `charmrun` command from the job script, the mother-superior assigns a unique port at

which the CCS-server must listen by appending the highlighted code to the user's original command as shown below:

```
> charmrun +p8 ./exec ++server ++server-port=1234
```

Users are not permitted to manually activate the CCS-server. This allows the mother-superior to assign unique port numbers to all Charm++ applications that may run on the same space-shared node. Figures 5.1 and 5.2 illustrate the steps of an expand and shrink process in the TORQUE RMS, respectively. When the scheduler initiates an expand operation, it sends the new list of hosts to the `server`, which is to be added to the job. The `server` updates the internal information and forwards the list to the mother-superior executing the job. The mother-superior modifies the node list (*hostfile*) and performs a *dyn_join* operation to dynamically associate the new nodes with the job. It then sends the CCSExpand message through the CCS-client API to inform the application. The reply to this message from the CCS-server is immediate and the Charm++ runtime starts using these resources after the next synchronization point in the application (typically between iterations). A similar process is carried out during a shrink operation, except that after the mother-superior sends the CCSShrink message to the application, the reply is not immediate. The CCS-server replies only after the data from the nodes to be removed are retrieved and the processes are cleaned at the next synchronization point.

5.4 Scheduling Malleable Jobs with the Maui Scheduler

To support scheduling malleable jobs, we further extended the Maui scheduler in the following way:

- We enhanced the resource allocation mechanism to expand and shrink a resource allocation set
- We devised a *dependency-based expand/shrink* (DBES) algorithm for the efficient scheduling of malleable jobs
- We enriched Maui's iteration with the combined scheduling of rigid, malleable, and evolving jobs

All malleable jobs are always scheduled according to their minimum requirements and later expanded. The DBES algorithm consists of two expansion steps. The first expansion step, contrary to other strategies, which is based on analyzing job and resource dependencies, and targets increasing throughput. The earliest start time of a *StartLater* job (i.e., a job that can only be started at a later point of time) is the deadline of that running job after whose completion all the resources requested by the *StartLater* job become available.

Algorithm 3 Maui Iteration

```
1: while TRUE do
2:     Obtain resource information from TORQUE
3:     Obtain workload information from TORQUE
4:     Update statistics
5:     Refresh reservations
6:     Prioritize eligible static requests
7:     Prioritize eligible evolving requests
8:     Schedule static requests in priority order and create reservations (without job start)
9:     for each evolving request do
10:        Allocate idle resources
11:        if Not enough idle nodes are available then
12:            Shrink expanded malleable jobs to find resources
13:        end if
14:        if Enough idle nodes found then
15:            Apply fairness policies and determine if job expansion is allowed
16:            if Expansion is allowed then
17:                Allocate resources for evolving job
18:            else
19:                Reject the dynamic request
20:            end if
21:        else
22:            Reject the dynamic request
23:        end if
24:    end for
25:    Reschedule static requests and create reservations (with job start)
26:    Update job dependencies according to the new system state
27:    for each reserved job do
28:        Prioritize malleable jobs in the order: (i) malleable job expanded for this reserved
           job, (ii) malleable job expanded for no specific reserved job, (iii) malleable job ex-
           panded for other reserved jobs
29:        Analyze if expanded malleable jobs can be shrunk in the above order to make enough
           nodes available to start the reserved job
30:        if enough nodes were found then
31:            Shrink those malleable jobs that provide the required resources
32:            Allocate the resources to the reserved job and start the job
33:        end if
34:    end for
35:    Reschedule static requests and create reservations
36:    Update job dependencies according to the new system state
37:    for each reserved job do
38:        if job depends on one malleable job then
39:            Expand the malleable job with the available nodes
40:        else if job depends on more than one malleable job then
41:            Equipartition available resources among these malleable jobs
42:        end if
43:    end for
44:    Update job dependencies
45:    Backfill non-reserved static requests from the job queue
46:    Equipartition available idle nodes among other running malleable jobs
47: end while
```

For example, consider a four-node system with two running jobs A and B using one node each for a scheduled period of 1 hour and 1/2 hour, respectively. If a queued job C requires 3 nodes for execution, it can start as soon as B is completed because two nodes are already available. On the other hand, if C requires 4 nodes for execution, it has to wait longer until the completion of job A. In our approach, we determine the dependencies of all *StartLater* jobs in the order of their priority. If the job on which a *StartLater* job depends is malleable, it is expanded using the available resources up to its user-specified maximum. Jobs expanded in this step maintain information about the dependent job in the queue for which the expansion was made. The second expansion step targets improving resource utilization and fairness, thereby equipartitioning the available resources across malleable jobs.

The complete Maui iteration for the combined scheduling of rigid, evolving, and malleable jobs is shown in Algorithm 3. In the first step, all the static and evolving requests are prioritized separately (lines 6-7). Static requests are scheduled, which creates the *StartNow* (job that can be started immediately) and *StartLater* jobs (line 8). At this point (lines 9-24), *StartNow* jobs are not yet started. Evolving requests are now scheduled, which may steal resources from the *StartNow* jobs, thereby causing delay to the *StartNow* as well as to the *StartLater* jobs, as explained in Chapter 4.

When no idle nodes are available, the system determines whether shrinking expanded malleable jobs can serve the evolving requests. At this point, the malleable jobs expanded in the second expansion step are considered first. If not enough nodes can be extracted from these jobs, the other malleable jobs expanded during the first expansion step are considered. If sufficient nodes are found, the malleable jobs are instructed to release the nodes. The dynamic fairness policies are then applied again to determine whether the evolving request can be satisfied with the newly available nodes. If yes, the evolving job is granted these nodes. Otherwise, the nodes obtained from the shrink operation are used later for expansion of malleable jobs or backfilling. Note that the dynamic fairness policies are checked only after shrinking the jobs that have selected to provide nodes for the evolving request. If the dynamic fairness policies do not allow satisfying an evolving request, the nodes may be allocated back to the jobs from which they were released. In the future, we plan to improve the system to enable it to apply dynamic fairness policies without having to shrink the jobs so as to reduce the overhead.

After all the evolving requests have either been satisfied or rejected, a new schedule of static requests is produced as the state of the system and job dependencies may have changed (line 25). *StartNow* jobs produced at this step are started immediately. Any expanded malleable jobs maintaining invalid job dependencies are cleared (line 26). Now a shrink phase is initiated to attempt to locate nodes for *StartLater* jobs (lines 27-34). Starting from the *StartLater* job with the highest-priority, the scheduler analyzes whether shrinking expanded malleable jobs can yield enough nodes to start the *StartLater* job. Malleable jobs are considered for shrinking in the following order: (i) expanded malleable jobs on which the reservation of this *StartLater* job originally depended, (ii) malleable jobs expanded during the second expansion step (i.e., expanded for no specific *StartLater* job), and (iii) malleable jobs expanded for other *StartLater* jobs which have lower priority than this *StartLater* job. If enough nodes are found, the malleable jobs are instructed to release the required resources and are the nodes are allocated to

the *StartLater* job so that it can start immediately. This procedure is then repeated for every *StartLater* job. After this, the iteration proceeds to the next phase.

Since there might have been changes in the system state again (due to starting more jobs), the queued jobs are scheduled again to create `ReservationDepth` number of *StartLater* jobs, and job dependencies are recomputed (lines 35-36). The first expansion step is initiated where the computed job dependencies are used to expand the malleable jobs each *StartLater* job depends on, as explained above (lines 37-43). During this step, nodes can also be stolen from other malleable jobs that were (i) either expanded for no specific *StartLater* job (i.e., in the second expansion step) or (ii) expanded for a *StartLater* job that has lower priority than the currently considered *StartLater* job.

Such a transfer of nodes allows a malleable job to be expanded as much as possible to increase its speedup and allow the *StartLater* job to be started earlier. In some cases, a *StartLater* job may also depend on two malleable jobs having the same completion time. In such scenarios, the available resources are equipartitioned among these malleable jobs. Running malleable jobs on which no *StartLater* job depends are not expanded in this step. After the first expansion step, a backfill step is initiated, which ensures that only those jobs are started that will not delay any *StartLater* job (line 45). Finally, after the backfill step, a second expansion phase begins where the malleable jobs on which no other job depends are expanded with the available resources through equipartitioning (line 46).

One of the important differences between the proposed algorithm and other approaches is that it gives due importance to backfilling with a two-step expansion process. As mentioned in Section 2.2, other methods perform a shrink operation only if the next job in the queue cannot be started. This is followed by an expand phase where running malleable jobs are expanded. As a final step, backfilling is performed with nodes available after expansion. An approach that ignores backfilling [81] may be suitable for a workload with 100% malleable jobs but not for a workload that also consists of rigid jobs. In our approach, we perform a "needful" expansion, followed by backfilling and equipartitioned expansion. Also, in every iteration, dependencies are recomputed only if there is a change of state in the system, thereby avoiding unnecessary and frequent dependency computations. In the presence of evolving jobs, our approach attempts to its best to select those malleable jobs for shrinking that will least affect the throughput. Furthermore, since the number of *StartLater* jobs can be configured by the `ReservationDepth` parameter, administrators can modify it to control the behavior of the scheduler according to the site's workload characteristics. At a site with a large number of malleable jobs, the `ReservationDepth` can be increased to gain more from dependency-based expansion, while at a site with a generally low number of malleable jobs, it can be reduced to favor backfilling. The resources are charged only for the amount of time they are used. In the future, we also plan to provide administrator commands to manually shrink or expand jobs, which is useful for fault tolerance and easy proactive migration.

5.5 Evaluation

In this section, we evaluate the proposed batch system and analyze its performance with respect to throughput, system utilization, and overhead.

5.5.1 Experimentation setup

All the experiments were conducted on a 15-node cluster system equipped with 2 Intel Xeon X5570 processors per node running at 2.93 Ghz (8 cores per node). A separate 16th node was used as the headnode running the extended TORQUE version 4.1.0 and Maui version 3.3.1. For a fair comparison, all the experiments were performed with `ReservationDepth` in Maui set to 5. Common benchmark workloads for the evaluation of schedulers contain only rigid jobs. We are not aware of any benchmark with malleable or evolving jobs. Therefore, we modified the well-known ESP benchmark [46] to contain various percentages of rigid, malleable, and evolving jobs. The original ESP benchmark is composed of 230 jobs with 14 different job types. All the jobs run the same synthetic application. Each job type has a unique fixed execution time and uses a fraction of the total resources. To evaluate the DBES strategy, the synthetic application was replaced by a Charm++ mini-application called *LeanMD*. LeanMD is a Molecular Dynamics (MD) mini-application which performs a simplified version of the force calculations of NAMD [118], a widely used MD code. LeanMD uses two Charm++ object arrays: (i) *cells* - a collection of atoms in 3D space, and (ii) *computes* - perform force calculation on atoms. To comply with the benchmark, each LeanMD mini-application was executed with varying numbers of cells and iterations to fit the job type's running time. As an evolving job, a synthetic MPI application with an evolving pattern similar to the real-world application Quadflow [57] was introduced. Quadflow is an MPI-based CFD flow solver that solves the compressible Navier-Stokes equations using a cell-centered fully adaptive finite volume method on a locally refined grid. Our synthetic application imitates a typical high-enthalpy stagnation point problem of supersonic flow around a 2D Cylinder at Mach 5.28. The application evolves after 16% percent of its static runtime and requests 4 additional cores. If the resources are not available at that point, the job continues and requests resources again after 25% of the total static running time as a second chance to obtain resources. If both attempts fail, the job continues with the current allocation until its completion. However, if the evolving request was satisfied, a linear reduction of the execution time is assumed for the evolving job. Table 5.1 shows the various job types of the modified ESP benchmark, the fraction of cluster resources each job uses, the total number of jobs of each type, the static execution time of each job type, the number of cells used when the job type was converted to malleable and the execution time of the job if it's dynamic requests are granted when it evolves.

5.5.2 Scheduling malleable jobs

Figure 5.3 shows the comparison of the total execution time of the ESP workload with varying amounts of malleable and rigid jobs with the DBES and the other strategies discussed in

Table 5.1: Properties of all job types of the modified ESP benchmark. .

Job Type	Size of System	Count	Static Execution Time	Malleable [Number of Cells]	Evolving [Execution Time in Secs]
A	0.03125	75	267	6x6x6	-
B	0.06250	9	322	8x8x8	-
C	0.50000	3	534	15x15x15	-
D	0.25000	3	616	10x10x10	-
E	0.50000	3	315	15x15x15	-
F	0.06250	9	1846	8x8x8	1230
G	0.12500	6	1334	9x9x9	1067
H	0.15820	6	1067	11x11x11	896
I	0.03125	24	1432	6x6x6	-
J	0.06250	24	725	8x8x8	-
K	0.09570	15	487	7x7x7	-
L	0.12500	36	366	8x8x8	-
M	0.25000	15	187	10x10x10	-
Z	1.00000	2	100	4x4x4	-

Section 2.2. These are earliest started first (ESF), earliest deadline first (EDF), latest deadline first (LDF) and naive equipartitioning (EP) strategies. The *rigid* strategy executes the workload without any expand/shrink operations—irrespective of the number of malleable jobs present. It can be observed that the DBES strategy has a lower execution time in all cases compared to the other strategies. With 100% malleable jobs, the DBES strategy performs best with about 32% higher throughput than rigid scheduling and about 7% higher throughput than the best-performing state of the art strategy (in this case, EP). For large systems with longer workloads, this impact will be of higher magnitude. Furthermore, unlike other strategies, DBES is consistent in achieving the best total execution time. For example, while the equipartitioning strategy is the one that performs best among the state of the art strategies for a workload with 60% malleable jobs, it delivers the worst performance with 10% and 30% malleable jobs, and second worst with 90% malleable jobs. Similarly, the ESF strategy performs best amongst the state of the art strategies for a workload with 20% malleable jobs, but worst with 50% malleable jobs. In certain cases the other strategies can perform even worse than rigid scheduling. For instance, the EDF strategy with 20% malleable jobs and EP strategy with 30% malleable jobs took longer execution times than rigid scheduling. This is a direct effect of inefficient selection of malleable jobs for expansion and shrinkage. The DBES strategy never shows such a pattern as the dependency-based analysis ensures that malleable jobs are expanded only if this may facilitate an earlier start time for queued jobs. Otherwise, backfilling is given precedence to improve

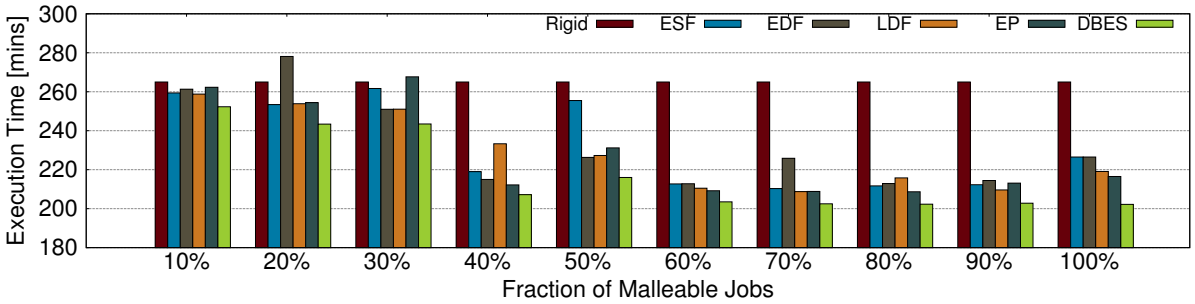


Figure 5.3: Time for completion of the modified ESP workload with varying amounts of rigid and malleable jobs.

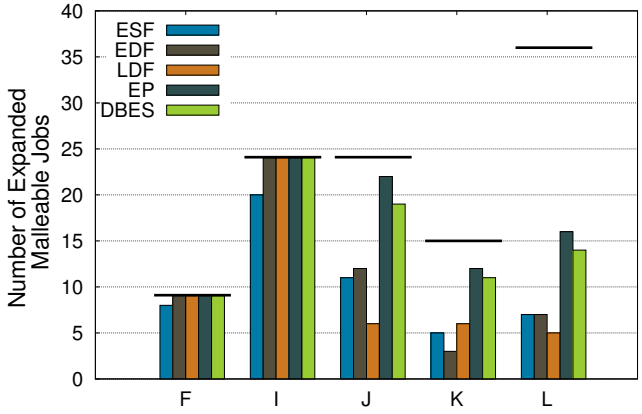


Figure 5.4: Comparison of the number of expanded malleable jobs belonging each category under various strategies for 50% malleable jobs. The total number of actual malleable jobs in each category is indicated by a horizontal line.

throughput. Thus, it extracts the best of malleability and backfilling, which other strategies fail to achieve. Overall, this implies that irrespective of the fraction of malleable jobs a site may have, the DBES strategy can be confidently applied to obtain the best throughput.

The reason behind the behavior of all these strategies and their resulting performance can be better understood by an in-depth analysis of expansion operations of all the strategies. As an example, Figure 5.4 shows the number of different types of malleable jobs expanded at some point in time during execution when run under each strategy. As ESF prefers to expand the job which started earliest, much of the expansion was made to type J jobs and it was not able to fully expand the long running F and I jobs which led to the smallest throughput. LDF preferred to expand jobs with long running times and therefore all F and I jobs were expanded at a very early stage. But this did not allow enough J, K and L jobs to be expanded to see a throughput gain. Similarly, along with F, I and J, EDF also expanded a few more L jobs but was not able to expand enough K jobs which actually have short running time. This was mainly due to the unavailability of idle resources when the majority of the K jobs were running. This was a result of backfilled A jobs using all the resources. As short jobs finished ahead of their walltime limit with more resources, they were used directly to start queued jobs instead of expanding running ones. Thus, due to a good use of resources, EDF delivers the best makespan amongst ESF, LDF, and EP albeit by expanding only a smaller number of malleable jobs. EP has the smallest makespan after ESF although it expanded the greatest number of malleable jobs

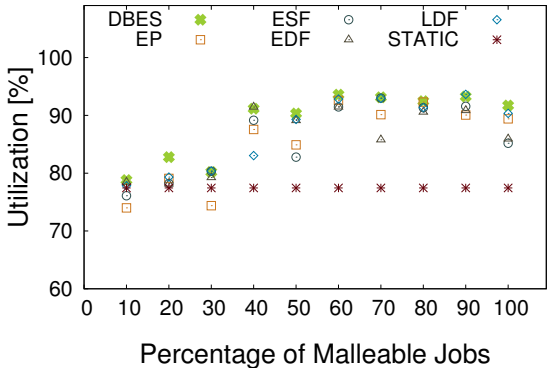


Figure 5.5: Comparison of the average system utilization achieved by all the strategies for the ESP workload with various percentages of malleable and rigid jobs.

of all the strategies. This is due to the equal distribution of resources and frequent expansion without giving priority to backfilling. We can observe that DBES has a similar expansion pattern to EP but still has about 7% higher throughput than EP. This is not only because it expands a reasonably large number of malleable jobs, but also because it does so in the right order and at an effective point in time while giving priority to backfilling when a gain cannot be obtained from expansion.

Figure 5.5 compares the overall average system utilization maintained by the strategies for workloads with various percentages of malleable jobs. It can be seen that in general the DBES strategy maintains the highest system utilization. On the other hand, other strategies also achieve average utilization close to or even slightly better than DBES in some cases, but still have lower throughput. For example, with 40% malleable jobs, the EDF strategy maintained a slightly better average system utilization than DBES but still had about 5% less throughput than DBES. Thus, DBES not only increases system utilization but also assures increased throughput. Note that the execution time of the workload with 50% malleable jobs was slower than that containing only 40%. This is because the workload for 40% malleable jobs was formed by making job types F, G, H, I, K, and L malleable, while the 50% was made with F, I, J, K, and L. The selection of job types to create malleable jobs was arbitrary in both the cases. The non-malleability of long running G and H jobs caused the longer execution time of the workload with 50% malleable jobs. Thus, the presence of larger numbers of malleable jobs does not always mean better performance than the presence of only a smaller number.

5.5.3 Combined scheduling of rigid, malleable, and evolving jobs

We demonstrate and analyze the combined scheduling of rigid, malleable and evolving jobs with an ESP workload containing 10% evolving (F, G and H), 40% malleable (I, J, K and L), and 50% rigid jobs. We are not aware of any other work that consists of a unified scheduling method for the combined scheduling of the above job types. Therefore, we combined our evolving job scheduling strategy with DBES and other strategies to compare their execution times. This is shown in Figure 5.6. We can see that DBES again has the fastest execution time with an increase in throughput of about 6% in comparison to the best performing state-of-the-art strategy (in this case, LDF). Table 5.2 presents the total number of evolving jobs that were satisfied

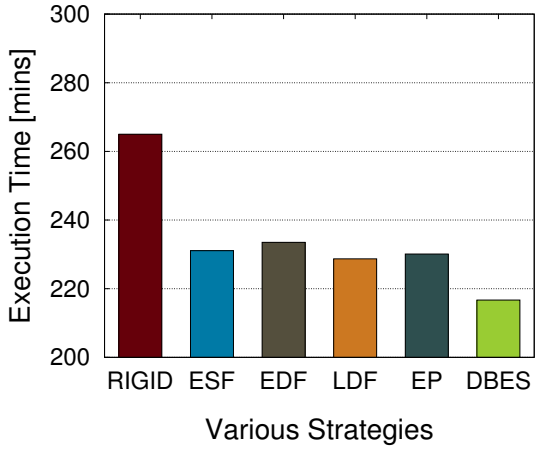


Figure 5.6: Time for completion of the modified ESP workload under various strategies with 10% evolving jobs, 40% malleable jobs and 50% rigid jobs.

Table 5.2: Comparison of the various malleable scheduling strategies when combined with evolving jobs.

	DBES	EP	ESF	EDF	LDF
Total no. of evolving jobs satisfied	11	11	13	17	12
Jobs shrunk to satisfy evolving jobs	9	9	9	10	8

in each case and the corresponding number of malleable jobs that were shrunk to obtain the resources for the evolving jobs. While EDF and ESF satisfied more evolving jobs, all strategies needed to shrink roughly an equal number of malleable jobs to obtain resources. This implies that a greater number of idle nodes were present in EDF and ESF during job evolution compared to other strategies. In other words, the inefficient system utilization was advantageously employed for the evolving jobs by the scheduler. Out of the nine malleable jobs that were shrunk in the DBES strategy, five malleable jobs had been expanded in the second expansion step (through equipartioning) and four malleable jobs had been expanded in the first expansion step (through dependency analysis). Thus, the improved performance is a combined result of the DBES strategy handling malleable jobs and the choice of malleable jobs for shrinking to make resources available for evolving jobs. The DBES strategy avoids as far as possible selecting malleable jobs expanded through the dependency analysis. Therefore, while almost the same number of evolving jobs were satisfied in all the strategies except EDF, DBES still achieves better performance.

5.5.4 Overhead

Figure 5.7 shows the overhead of expand and shrink operations. For expansion, the total time required to expand a single-node job by adding upto 14 extra nodes is plotted. Naturally, the

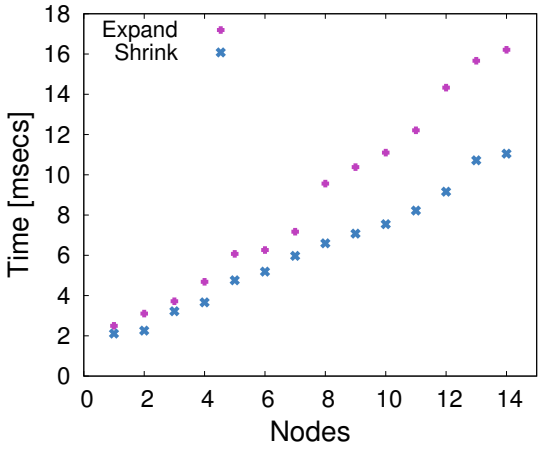


Figure 5.7: The time taken for (i) adding 1 - 14 additional nodes to a job initially using 1 node (expansion), and (ii) removing 1 - 14 nodes from a job initially using 15 nodes (shrinking).

time required for expansion increases with am increasing number of nodes due to communication with a larger number of nodes during the *dyn_join* operation. However, the time stays below 20 milliseconds for an expansion upto 14 additional nodes, which is fast and efficient. For shrinking, the plot shows the total time required for *immediately* removing 1 to 14 nodes from a job that initially used 15 nodes. By "immediately", we mean that nodes are released instantly after receiving a shrink message. The total time taken for such an operation increases with a larger number of nodes to be removed but remains in the milliseconds range. This is generally faster than expansion since *dyn_disjoin* communicates much less data. However, the time taken for a shrink operation depends on the time required for the task running on the nodes to be removed to complete. Since the shrink message can be initiated at any time, the time required for the task to be completed cannot be predicted beforehand. Therefore, in reality shrinking usually takes longer than expansion. In the future, we plan to extend the communication mechanism to also include minimal application feedback in order to initiate shrink messages at a convenient point in time so as to reduce the waiting time until task completion.

5.6 Summary and Conclusion

As we move towards the next generation of supercomputers, adaptivity is expected to gain more importance so as to improve fault tolerance, increase energy efficiency and explore new application domains. As programming models become more adaptive in nature, malleability, a long desired property, could emerge as a natural by-product. Thus, adaptive resource management and scheduling is essential to gain high throughput and faster response times for a workload of adaptive jobs.

In this contribution, we propose algorithms for scheduling rigid, malleable, and evolving jobs in combination. The novel malleable scheduling strategy called DBES expands and shrinks malleable jobs based on dependencies between job reservations. The technique is combined with backfilling to gain best performance for varying dynamics of the workload. Furthermore, equipartitioning is applied for fairness with resources that remain unused under both dependency-based expand/shrink and backfilling. Results show that DBES demonstrates con-

sistently superior makespan and throughput in comparison to other state of the art scheduling strategies. The results also show that it is the best strategy to be applied together with scheduling unpredictably evolving applications.

As malleability is expected to flourish widely, it will attract more research in scheduling and resource management as its usage trends will vary in different ways. The proposed methods are not only better than existing techniques but also easily extendable to suit upcoming scenarios. For example, minimal application feedback can be established and dependency-based malleable job scheduling with feedback considerations can be employed. In order to achieve better parallel efficiency, expansion and shrinkage based on application scaling patterns is worth exploring as well. Thus, this contribution lays the foundation for a wide scope of future research in this field.

6 Fault Tolerance

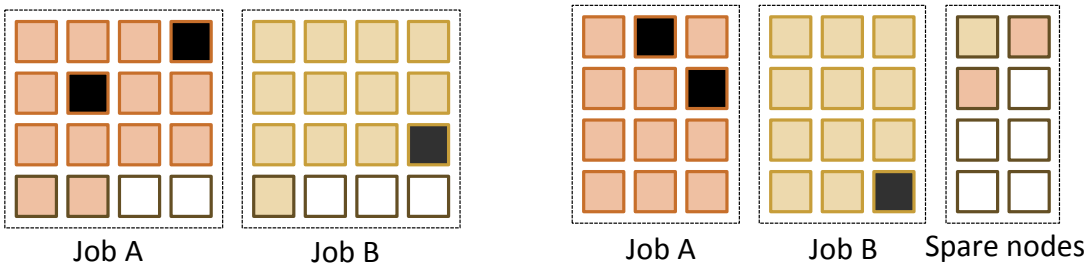
This chapter describes the concept of dynamic resource management for fault tolerance. It provides an overview of the approach, highlighting the motivation and benefits, before describing the dynamic node replacement algorithm in detail, followed by the experimental evaluation. This chapter is based on the master thesis of Marcel Neumann [119], which was performed under the supervision of the author of this dissertation. Most of the figures used in this chapter have been reproduced from the above mentioned thesis.

6.1 Dynamic Resource Management and Node Replacement for Fault Tolerance

As we move into the exascale era, fault tolerance has been conceded as the most important challenge, owing to the high failure rates that the exascale systems are expected to have. The mean time between failures (MTBF) is expected to be less than an hour for an exascale system. Although checkpoint/restart is a popular technique in petascale systems, existing methods cannot be directly applied on exascale systems.

In current practice, a running application is checkpointed on a regular basis. Checkpointing is initiated either by the application itself or the batch system. When the job is affected by node failure, batch systems cancel the job and restart the job from the latest checkpoint on a fresh allocation of nodes. Checkpoints are usually made to disk-based storage and are therefore time consuming. Applications that need 30 minutes per checkpoint are not uncommon. If the checkpoint time is close to the MTBF, then most of the time is spent on checkpointing and restarting with little application progress. This also lengthens job execution and turnaround time, which affects the overall throughput and availability of the system. An effective way to reduce the checkpoint time is in-memory checkpointing [120] and multilevel checkpointing such as FTI [94] and SCR [95]. Multilevel checkpointing involves combining different storage technologies pertaining to multiple levels of the storage hierarchy to store a checkpoint. The first few levels of storage are in-memory and remote-memory storage. It supports process failures and multi-node failures with a limit on the maximum number of nodes that can fail at a time. The last level of checkpointing is the file system. Despite the reduced checkpoint/restart time facilitated by this approach, the static resource management of current batch systems adds other significant overheads.

When a job is affected by node failure, static allocation mechanisms force the job to be cancelled, resubmitted and restarted on a new set of nodes. This introduces additional overhead. Even if the new set of nodes consists of a subset of the nodes previously used by the job (before failure), the processes and the data must be inserted afresh into the memory. Thus, the advantage of multilevel checkpointing is reduced. One way of circumventing this problem is to allocate dedicated spare nodes for each job, beyond what is actually required by the application.



(a) Static allocation: failed nodes replaced by statically allocated spare nodes

(b) Dynamic allocation: failed nodes dynamically replaced by nodes from a shared pool of spare nodes

Figure 6.1: Static and dynamic allocation of spare nodes in the event of node failure.

In the event of a node failure, these spare nodes can be put to use immediately without having to resubmit the job. The application can then be restarted using the data from in-memory checkpoints. However, this approach requires a relatively large number of spare nodes to stay prepared for faults. This leads to a significant amount of nodes to remain unused and results in poor system utilization and throughput. Furthermore, a seemingly sufficient number of spare nodes allocated for a job may still turn out to be inadequate, as the pattern of faults may drastically vary. When no spare nodes are available, the job has to fall back to requesting a fresh allocation through resubmission. Hence, static resource allocation mechanisms cannot be effectively used for fault tolerance at exascale.

In this context, dynamic resource-management facilities enable on-the-fly replacement of failed nodes by other healthy nodes. This approach has several advantages compared to traditional resource management. Figure 6.1 illustrates static and dynamic replacement of spare nodes during node failures.

1. **No resubmission overhead**
   Dynamic replacement eliminates the overhead of resubmitting and restarting the job. The job (the other running processes) can be paused until the failed nodes are replaced and the job can continue execution after the restarted processes are in a consistent state with the other processes.

2. **More efficient use of resources**
   Dynamic node replacement eliminates the necessity of allocating spare nodes to each job. This avoids resource wastage at the expense of a user's allocation account. Rezaei and Mueller [121] inferred that having a dynamic pool of nodes can reduce the required number of spare nodes by an order of magnitude as compared to attaching spare nodes to each job.

3. **Less full job restarts required**
   Under static allocation, the job has to be restarted completely after every node failure when there are no statically allocated spare nodes available. That is, all the healthy processes are also killed and restarted on a fresh allocation. For example, when one node

of a 64-node job fails, all the processes belonging to the other 63 nodes also need to be restarted. Restarting an application not only involves the overhead of spawning process but also loading data from the disk to the memory of all the processes, which involves sharing the network bandwidth from storage to the compute nodes. Therefore, the time taken for restarting an application increases with the number of processes to be restarted. With dynamic node replacement, a partial restart of a job is always sufficient. Only the failed processes need to be restarted even if multiple nodes fail incessantly. The active processes only need to rollback to a checkpoint state, thereby avoiding the time needed to spawn new processes and potentially loading data from the disk (especially when multi-level checkpointing is used as explained below). Therefore, the time for restart is greatly reduced. Since the old and restarted processes come to a consistent checkpoint state, messages lost in the process of going through a failure are resent and do not cause inconsistency.

4. **Better use of in-memory checkpoints and reduced frequency of disked checkpoint/restart**
   With dynamic node replacement enabled, full advantage of multilevel checkpointing can be realized. Single-node and multi-node failures (upto a certain number limited by the checkpointing method) can be recovered with in-memory checkpoints. Since the risk of a large number of nodes of a job failing at the same time is generally low, this approach would require only a low frequency of regular checkpoints to the disk.

5. **Improved resource availability**
   Dynamic node replacement can take full advantage of dynamic resource management. Since dynamic resource management supports malleable and evolving jobs, the batch system has a wider choice of obtaining idle nodes for replacement. For example, in the absence of idle nodes in the cluster, a malleable job can be shrunk to instantly replace a failed node.

Thus, dynamic node replacement is an important aspect for fast application recovery in current and future systems. Therefore, this work proposes a comprehensive scheduling algorithm for fault tolerance aimed at facilitating fast node replacement and high throughput even under high failure rates. The focus of this work is restricted to the functionality of the batch system. A complete recovery of the application requires either a checkpoint/restart framework or the parallel runtime of the application to perform the necessary actions, which is out of the scope of this work.

6.2 Dynamic Node Replacement Algorithm

The node replacement algorithm is designed to be a supplement to the main job scheduling algorithm or the *base scheduling algorithm*. The node replacement algorithm is invoked only in the event of node failure. As the base scheduling algorithm we use the scheduling algorithm introduced in Chapter 5, which schedules jobs in a FCFS fashion with backfilling and

DBES support. Moldable jobs are scheduled using the Supercomputer AppLeS (SA) described in Chapter 2.

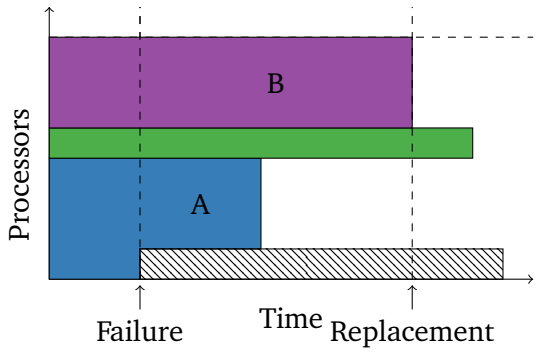
The node replacement algorithm essentially tries to assign replacement nodes to jobs affected by failure. When the node replacement algorithm is invoked, it cancels all job reservations made for the future and treats the affected jobs with highest priority. The scheduler collects the list of jobs that are affected by failure and attempts to replace the failed nodes for each affected job through the following options, in the order of presentation. Note that the waiting time incurred by the job until the replacement takes effect is not ascribed to its computation time and the job walltime is adjusted accordingly.

1. Local shrink
2. Use of idle processors
3. Remote shrink
4. Restarting moldable jobs
5. Waiting for processors to be come idle

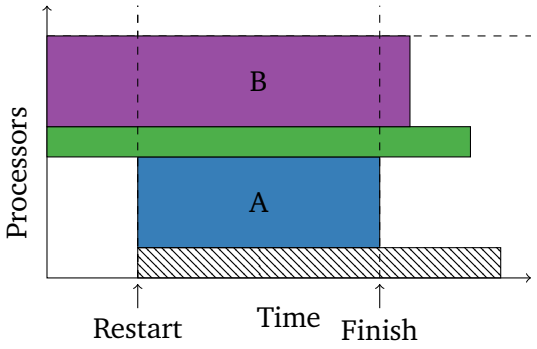
Local shrink. The first option is considered only for malleable jobs, wherein the scheduler attempts to shrink the job and continue execution by removing the failed nodes out of the job's allocation. This can be successful only if at least the minimum number of nodes to execute the job remain allocated to it after it has been shrunk. This avoids replacement while maintaining the conditions requested during the malleable job's submission, and therefore is a faster solution. The job scheduling algorithm can later expand the job again according to the DBES principle.

Use of idle processors. When the first option cannot be applied, the scheduler attempts to replace the failed nodes with idle ones as the second option. Idle resources may be found from the cluster's regular partition or a spare pool of nodes as described in Section 6.1, with priority given to the spare pool.

Remote shrink. When the first two options are unsuccessful, the scheduler considers as the third option replacing failed nodes with healthy nodes obtained by shrinking running malleable jobs. Malleable jobs that have been previously expanded with additional nodes are considered for this purpose. In order to maintain the DBES decisions towards better throughput, priority of malleable jobs selected for shrinking follows the order of malleable jobs expanded in the second stage of DBES (where resources are equipartitioned among malleable jobs) followed by malleable jobs expanded in the first stage of DBES (where resources are distributed to malleable jobs based on the dependencies in the reservation flow). The shrunk malleable jobs may later be expanded again by the DBES algorithm if sufficient nodes are available. Note that during this step of the node replacement algorithm, a combination of option 3 and option 2 is also considered.



(a) Job A is affected by a node failure. The earliest time for a possible replacement is when job B terminates



(b) The algorithm decides to restart job A on a smaller resource set, which leads to a completion time that is earlier than if it had waited for job B to terminate.

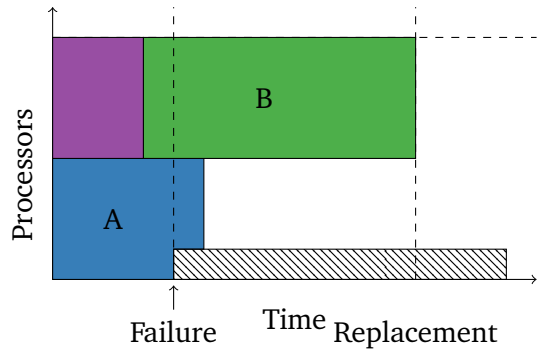
Figure 6.2: An example for the local restart of a moldable job.

Restarting moldable jobs. The fourth option considers two special cases only encountered with moldable jobs. In the first case, the affected job itself is a moldable job which can be restarted with a valid lower number of nodes (typically specified by the user during job submission). During failure, the latest point in time at which sufficient replacement nodes can be availed for the affected job can be determined using the job walltime information. Depending on this waiting time and the runtime estimates of the moldable job's request alternatives, restarting the moldable job with a lower number of nodes can lead to an earlier time of completion than waiting for replacement. Figure 6.2 exemplifies this scenario where a moldable job A is affected by a node failure. The algorithm restarts the affected moldable job on a reduced set of nodes rather than waiting for job B to terminate, leading to an earlier completion time.

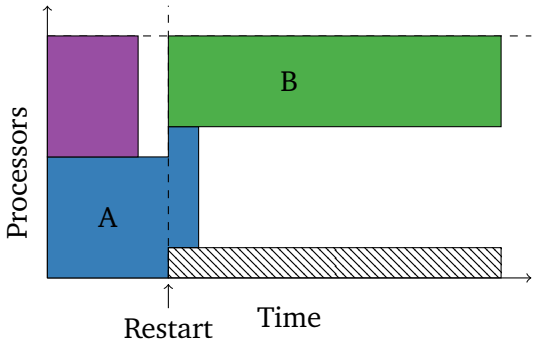
However, this approach may not yield the best result in certain situations. For example, if the affected moldable job has almost reached its completion, although restarting the job on a lower number of nodes may provide a better throughput than waiting for a replacement, it drastically increases the turnaround time of the job. Therefore, for such scenarios and for scenarios where the affected job is not a moldable job, the second case is considered, which attempts to restart other running moldable jobs with a lower number of nodes in order to free up nodes for replacement. This is illustrated in Figure 6.3. Job A is affected by a node failure when it has almost reached the walltime. However, its waiting time for replacement is long in contrast to the extra time which job B, that just started, would need if it was restarted on a reduced set of processors. The algorithm therefore decides to restart job B. Thus, the problem here is to determine the set of moldable jobs that must be restarted to serve the recovery needs of the affected job while minimizing the cost to free enough nodes. This is formally defined as follows.

We define a variable $x_{j,c}$ for every job $j \in J$ and their allowed job sizes $c \in C_j$ such that

$$x_{j,c} = \begin{cases} 1 & \text{if job } j \text{ should restart on } c \text{ processors} \\ 0 & \text{otherwise.} \end{cases} \tag{6.1}$$



(a) Job A is affected by a node failure. The earliest time for a possible replacement is when job B terminates.



(b) The algorithm decides to restart job B on a smaller resource set as its delay in completion time is smaller than if job A had waited for job B to terminate.

Figure 6.3: An example for restarting a remote moldable job.

The cost of restarting job j on c processors in terms of the delay in the completion of the job is given by:

$$r_{j,c} = x_{j,c}(e_j(c) - \bar{s}_j), \tag{6.2}$$

where $e_j(c)$ gives the completion time of job j when restarted instantly on c processors and $\bar{s}_j$ is the guarantee given by the scheduler for the completion time of job j when it was started for the very first time. Since the main aim is to minimize the delay in completion time for every job, this definition prevents the same job being considered multiple times for a restart. When the current job size is a_j for every job $j \in J$ and the number of processors needed for the failed job to recover is d, the values for $x_{j,c}$ can be obtained by solving the following integer programming problem:

$$\begin{aligned} \min_{x_{j,c}} & \sum_{j \in J} \sum_{c \in C_j} x_{j,c}(e_j(c) - \bar{s}_j) \\ \text{such that} \quad & \sum_{j \in J} \sum_{c \in C_j} x_{j,c}(a_j - c) \geq d \\ \text{where} \quad & \sum_{c \in C_j} x_{j,c} \leq 1 \quad \forall j \in J \\ \text{and} \quad & x_{j,c} \in \{0,1\} \quad \forall j \in J, c \in C_j \end{aligned} \tag{6.3}$$

Note that J consists of only all running moldable jobs including the affected job if it is moldable. Also, C_j does not have to include a_j for every $j \in J$. Job j is not restarted when $x_{j,c}$ is zero for every $c \in C_j$. The integer programming problem can then be solved using, for example, a branch-and-bound algorithm. This provides a solution $x_{j,c}$ for every job. The dynamic node replacement algorithm considers restarting jobs for which $x_{j,c} = 1$ when the delay in completion time of the failed job is less than the time that it will spend waiting for processors to be released by normal termination of other running jobs.

Waiting for processors to become idle. Finally, if all of these options are not able to provide an efficient replacement to the affected job, the fifth and the last option is considered, which is waiting for the completion of other jobs to acquire resources. In this case, the affected job is put at the top of the queue of pending jobs. As soon as nodes are available, they are allocated to the affected job.

6.3 Evaluation Environment

In this section, we present the evaluation environment used to study the dynamic node replacement algorithm. We first discuss the implementation of the algorithm for evaluation purposes. This is followed by the detailed description of two key components of the evaluation environment: workload model and failure model. The workload model generates the workload with the required characteristics such as runtime of each job, size of the workload and the job mix. The failure model generates the failure rate and pattern of node failure.

6.3.1 Implementation

The dynamic node replacement algorithm and the base job scheduling algorithm were implemented and evaluated on the custom discrete-event simulator based on the SimJava2.0 package [122]. In a discrete-event simulation, the state of a system changes at discrete points in time according to the occurrence of certain *events*. In this context, the events that change the state of our system are: job arrival, job completion, node failure and node recovery. A set of variables define the state of the system at a given point in time. The variables of importance to this context are: job queue, the list of jobs currently in execution, and the list of jobs that are affected by node failures. Each event is processed with the respective action after updating the state of the system. For example, a node failure event moves all jobs running on a failed node to the list of affected jobs. This is followed by a call to the dynamic node replacement algorithm to attend to the affected jobs. The workflow of the simulation is shown in Figure 6.4. The base resource mapping and backfilling algorithm is used from the implementation provided by the GridSim toolkit [38]. For solving the linear programming problem defined in the node replacement algorithm, we use the GNU Linear Programming Kit [123].

6.3.2 Workload model

Although there are many real workloads and workload models publicly available [124], they only reflect rigid jobs. We required workload models that already include or enable easily integrating moldable and malleable jobs. Therefore, we chose two models for rigid and moldable jobs proposed by Cirne [85]. These models were developed by Cirne after studying the workload traces listed in Table 6.1. Malleable jobs were generated by extending the moldable job model.

For the purpose of evaluation, we generate workloads of required size and mix of job types from the SDSC workload trace for a given system size. The jobs in the workload are character-

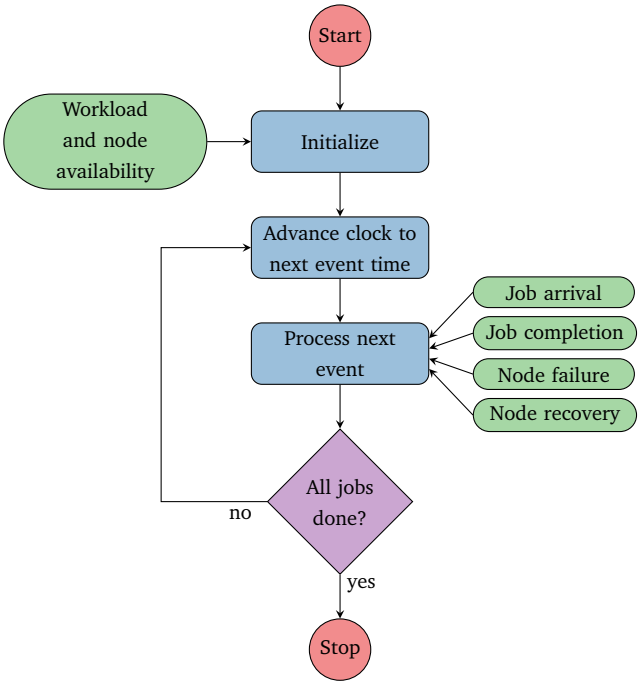


Figure 6.4: Workflow of the discrete-event simulation.

Table 6.1: Workload traces used by Cirne [85].

Name	Machine	Processors
ANL	Argonne National Laboratory	120
CTC	Cornell Theory Center SP2	430
KTH	Swedish Royal Institute of Technology SP2	100
SDSC	San Diego Supercomputer Center SP2	128

ized by the job and speedup models explained below. The speedup model is mainly used by the job models to characterize the job runtime for the intended system size.

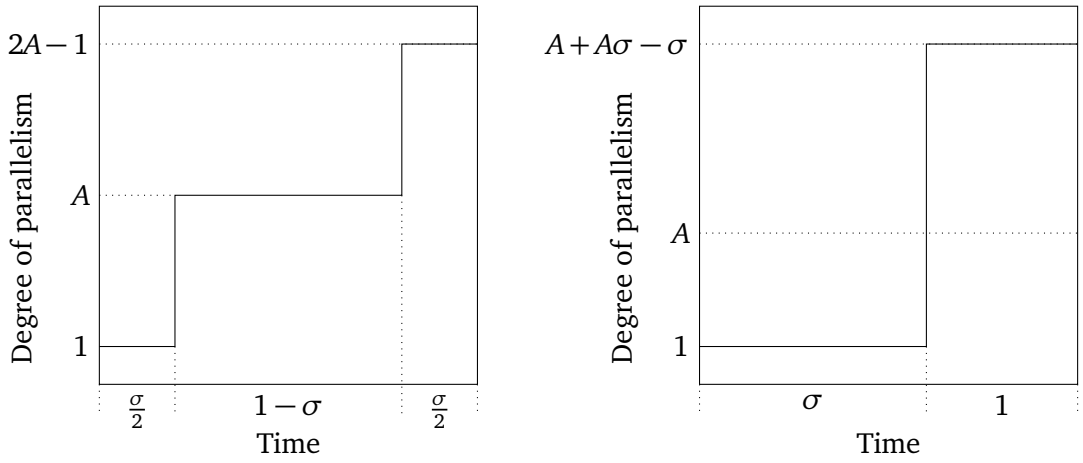
6.3.2.1 Speedup model

Since our methods use moldable and malleable jobs, it was essential to use a speedup model to capture the flexibility of the jobs according to their types. The speedup is typically defined as how much faster a job executes on n processors compared to its serial execution on only one processor. The speedup function S for a job running on n processors is given by:

$$S(n) = \frac{T(1)}{T(n)}, \tag{6.4}$$

where $T(n)$ denotes the job's runtime on n processors. The theoretical maximum speedup of a job is always limited by the fraction f of the computer program that has to be strictly serial due to algorithmic constraints. This is known as Amdahl's law [125] and is given by:

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{f + \frac{1}{n}(1-f)} < \frac{1}{f} \text{ if } n = \infty, \tag{6.5}$$



(a) Low-variance model. (b) High-variance model.

Figure 6.5: Hypothetical parallelism profiles proposed by Downey [89].

where $f \in [0,1]$ denotes the fraction of the computer program that is strictly serial.

Downey proposed a speedup model that constructs a hypothetical parallelism profile to reflect the behavior of common parallel applications [89]. For a job, this profile is defined by two parameters:

$A =$ average parallelism

$V =$ variance in parallelism

The average parallelism of a job is defined as the average number of busy processors during its execution when the system provides an unlimited number of processors [126]. With respect to variance in parallelism, the speedup model distinguishes between parallelism profiles with low variance and parallelism profiles with high variance. For a job running on n processors with a low variance in parallelism, the speedup function S is given by:

$$S(n) = \begin{cases} \frac{An}{A+\sigma(n-1)/2} & \text{if } 1 \le n \le A \\ \frac{An}{\sigma(A-1/2)+n(1-\sigma/2)} & \text{if } A \le n \le 2A-1 \\ A & \text{if } n \ge 2A-1, \end{cases} \tag{6.6}$$

where $\sigma \in [0,1]$ is an approximation of the coefficient of variance in parallelism. For such a job, except for some fraction σ of the job's duration, the degree of parallelism is equal to A. This fraction σ is divided equally into two parts: a part of high parallelism and a part of serial execution. An example is shown in Figure 6.5(a).

For a job running on n processors with a high variance in parallelism, the speedup function S is given by:

$$S(n) = \begin{cases} \frac{nA(\sigma+1)}{A+A\sigma-\sigma+n\sigma} & \text{if } 1 \le n \le A+A\sigma-\sigma \\ A & \text{if } n \ge A+A\sigma-\sigma, \end{cases} \tag{6.7}$$

where $\sigma \in \mathbb{R}^+$ is unbounded. Figure 6.5(b) shows an example of such a profile. It can be observed that the when $\sigma = 1$, the speedup functions are identical. The speedup curves for

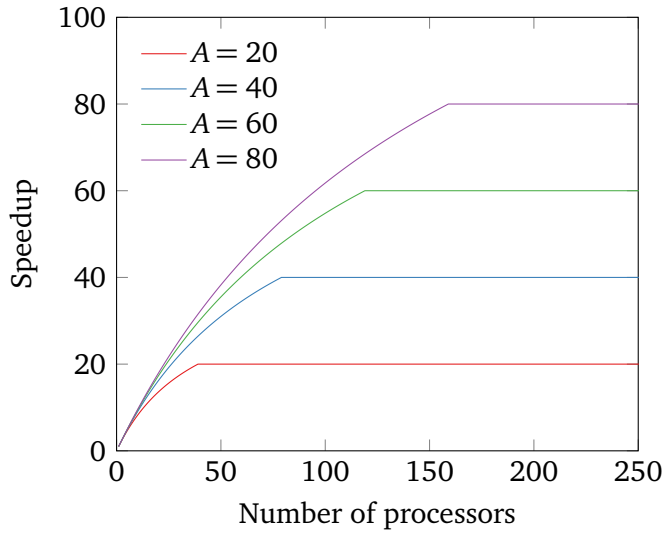


Figure 6.6: Downey's speedup curves for varying values of A when $\sigma = 1$ is fixed.

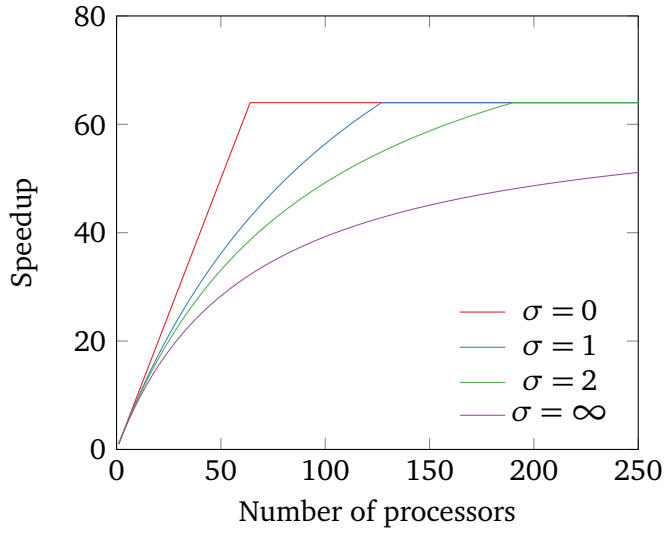


Figure 6.7: Downey's speedup curves for varying values of σ when $A = 64$ is fixed.

varying values of A when $\sigma = 1$ are shown in Figure 6.6. It visualizes that the average parallelism A determines the maximum speedup that can be achieved by a job. Similarly, the speedup curves for varying values of σ when A are fixed as 64 is shown in Figure 6.7. Thus, how fast a job reaches its maximum speedup is determined by the variance in parallelism. A job with small variance in parallelism is closer to linear speedup than a job with high variance in parallelism.

6.3.2.2 Rigid job model

Downey observed that the distribution of job sizes follows approximately a log-uniform distribution [89], which was later confirmed by Cirne [85]. The cumulative distribution function of a log-uniform distribution is given by:

$$\mathrm{CDF}(x) = \chi \log_2(x) + \rho, \tag{6.8}$$

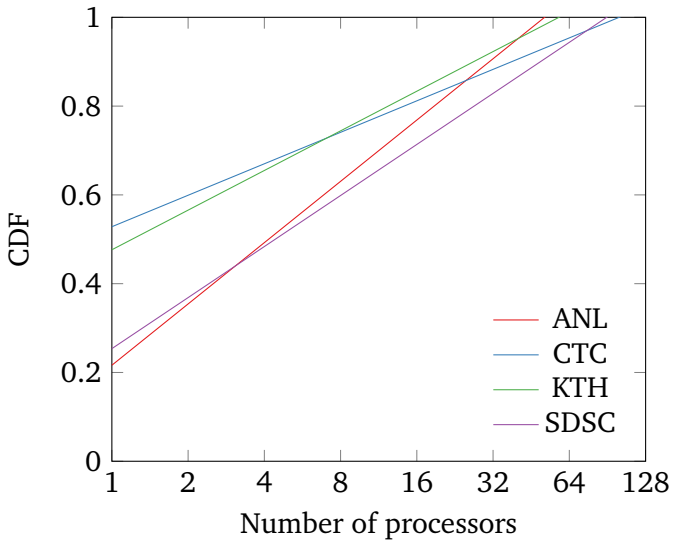


Figure 6.8: CDF for log-uniform distribution of job sizes [85].

Table 6.2: Summary of parameters for the rigid job model [85].

Characteristic	Model	Parameters
Job size	Log-uniform distribution	$\chi = 0.12$, $\rho = 0.20$
Power-of-2 job sizes	Probability p	$p = 0.75$
Estimated runtime	Log-uniform distribution	$\chi = 0.10$, $\rho = -0.75$
Accuracy a	Gamma distribution	$\alpha = 0.6$, $\beta = 0.6$

where χ and ρ are the slope and the intercept of the line in the log space.

The distribution of job sizes for the four reference workloads are shown in Figure 6.8. With respect to the rigid job runtimes, Cirne [85] defined the accuracy a of a job request to be the fraction of time that is eventually used by a supercomputer job and it is given by:

$$a = \frac{T(n)}{T_e(n)} \tag{6.9}$$

where $T(n)$ is the actual runtime of a job with size n and $T_e(n)$ is the runtime estimate or the walltime of the job. Since job walltime estimates are important for scheduling decisions, this describes the ability of a user to provide a good walltime estimate. Thus, by modeling two of these three parameters, the third one can be derived. Cirne [85] modeled the walltime estimate and accuracy of a job, leading to the distributions of walltime estimates for the four reference workloads (shown in Figure 6.9) and distributions of the accuracy (shown in Figure 6.10). These parameters have been adopted in our evaluation and the summary of these parameters is shown in Table 6.2.

6.3.2.3 Moldable job model

A moldable job consists of a set of a rigid requests with each request having a job size n, a runtime $T(n)$, and a runtime estimate or walltime $T_e(n)$. For each job size, the speedup

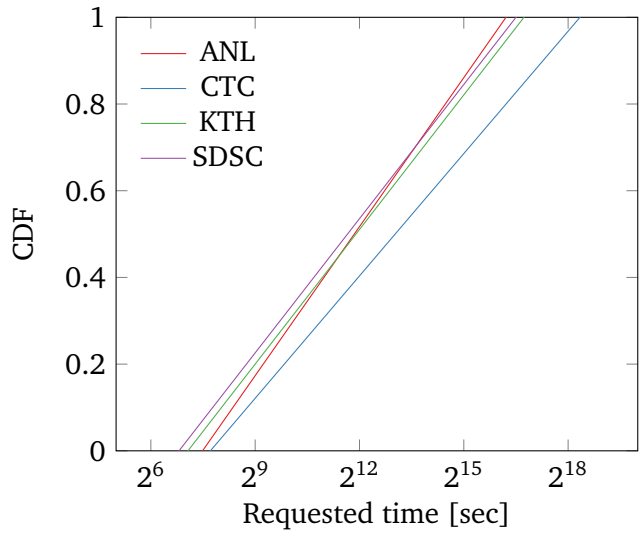


Figure 6.9: CDF for log-uniform distribution of runtime estimates [85].

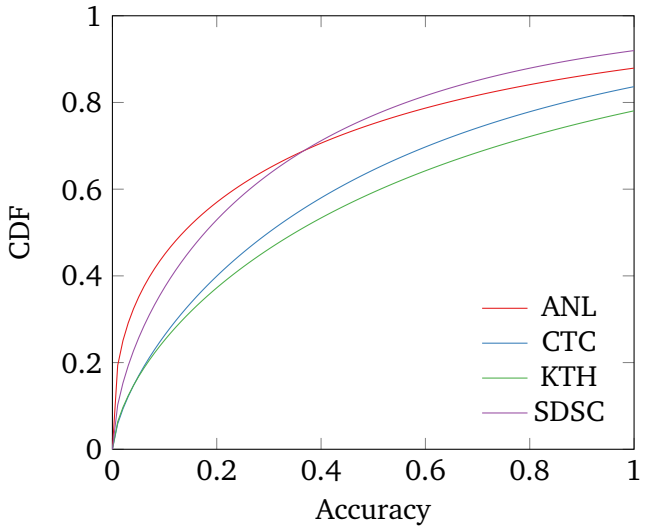


Figure 6.10: CDF for gamma distribution of runtime estimation accuracy [85].

model can be used to determine the job runtime and Equation 6.9 can be used to determine the walltime. Cirne generated job sizes by setting a minimum and a maximum job size and selecting a set of sizes within this range [85]. Cirne also conducted a survey to derive the distributions of the minimum job size and the number of requests a user is willing to provide. These are shown in Figure 6.11 and Figure 6.12 respectively. To determine the maximum size c_{max} of a job, Downey's model was used and is given by:

$$c_{max} = \begin{cases} 2A-1 & \text{if } \sigma \leq 1 \\ A+A\sigma-\sigma & \text{if } \sigma \geq 1. \end{cases} \tag{6.10}$$

In order to apply the speedup model for each job size, it is essential to determine the parallelism profile of the job by in turn determining the average parallelism and its variance. Cirne modeled the average parallelism after a survey question asking for an efficient job size, which is the size of a job for which the rectangle in $processors \times time$ space is minimal. Note the

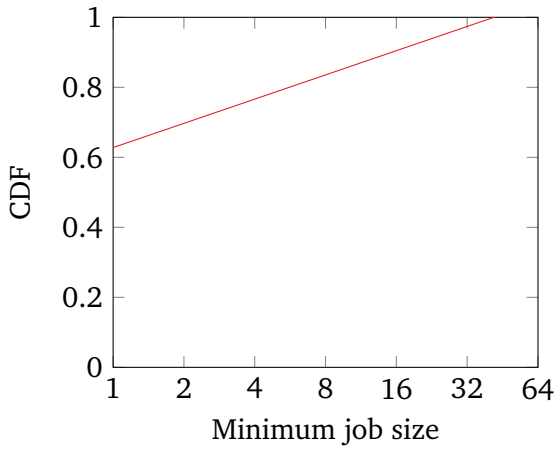


Figure 6.11: CDF for log-uniform distribution of minimum job size [85].

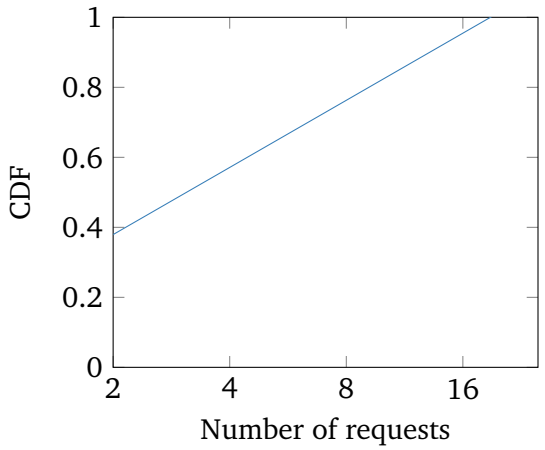


Figure 6.12: CDF for log-uniform distribution of number of requests [85].

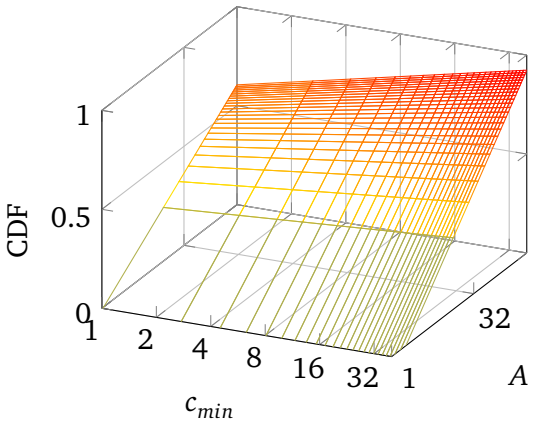


Figure 6.13: CDF for joint log-uniform distribution of A and c_{min}.

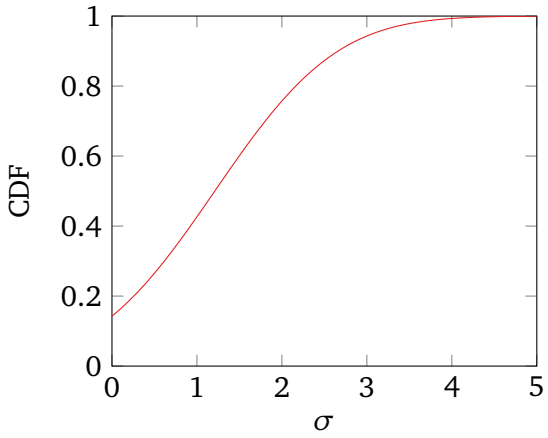


Figure 6.14: CDF for normal distribution for coefficient of variance in parallelism σ [85].

possible job sizes for the job is restricted by the application's memory requirements, amount of parallelism and algorithmic constraints. Their distribution is described by a joint log-uniform distribution and is given by:

$$\mathrm{CDF}(x,y) = \varphi \cdot \log_2(x) \cdot \log_2(y) + \gamma \cdot \log_2(x) + \eta \cdot \log_2(y) + \rho. \tag{6.11}$$

The distribution is shown in Figure 6.13. Cirne also estimated the coefficient of variance in parallelism after asking two survey questions for the most efficient job size s_e and the maximum job size s_{max}. This is given by:

$$\sigma = \frac{s_{max} - s_e}{s_e - 1}. \tag{6.12}$$

The distribution of σ is described by a normal distribution which is shown in Figure 6.14. These parameters are used in our evaluation and a summary is shown in Table 6.3.

Table 6.3: Summary of parameters for the moldable job model [85].

Characteristic	Model	Parameters
Minimum job size	Log-uniform distribution	$\chi = 0.06920$, $\rho = 0.6279$
Number of user requests	Log-uniform distribution	$\chi = 0.1918$, $\rho = 0.1876$
Average parallelism	Joint log-uniform distribution	$\varphi = 0.009548$, $\gamma = -0.01877$
		$\eta = 0.07468$, $\rho = -0.009198$
Variance in parallelism	Normal distribution	$\mu = 1.209$, $\sigma = 1.132$

6.3.2.4 Malleable job model

Malleable jobs are submitted with the same parameters as described in Chapter 5. For describing these jobs, the same parameters of the moldable job model explained in Section 6.3.2.3 are used. Additionally, instead of allowing only a set of job sizes within the minimum and the maximum job-size range, a malleable job can accept all the sizes within that range.

6.3.3 Failure model

A failure model describes the occurrence of failures in a supercomputer. This can be modeled by studying a variety of failure traces that are publicly available [127]. A study by Schroeder and Gibson [128] showed that it is difficult to describe a system after the first year of installation using a common probability distribution. The remaining years can be described using a Weibull distribution. Its cumulative density function is given by:

$$\mathrm{CDF}(x) = \begin{cases} 1 - e^{-(x/\lambda)^k} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0, \end{cases} \tag{6.13}$$

where α is its scale and k is its shape. Depending on the value of the shape parameter k, its failure rate is either decreasing ($k < 1$), constant ($k = 1$), or increasing ($k > 1$). Schroeder and Gibson observed that the mean time to repair can be well described by a log-normal distribution. Its cumulative density function is given by:

$$\mathrm{CDF}(x) = \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\left[\frac{\ln x - \mu}{\sqrt{2}\sigma}\right] \tag{6.14}$$

where μ is the mean, σ is the standard deviation, and $\mathrm{erf}(z)$ is the error function defined by:

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \exp^{-t^2}\, dt. \tag{6.15}$$

For the purpose of evaluation, we chose the mean time to interrupt for the system as 1 hour, which has been estimated to be so for an exascale system [129]. The distribution is shown in Figure 6.15. The distribution for time to recover from a failure is shown in Figure 6.16 and a summary of the parameters is shown in Table 6.4.

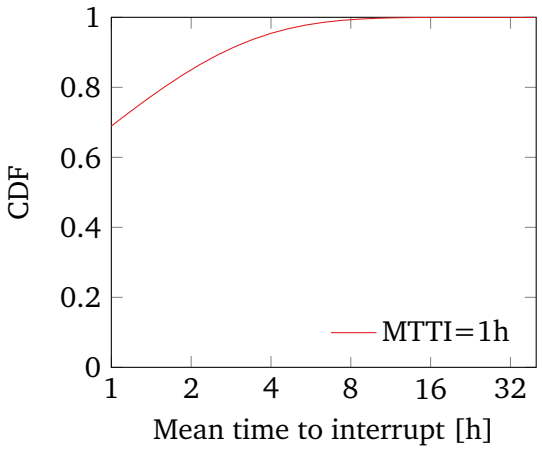


Figure 6.15: CDF for Weibull distribution of system MTTI.

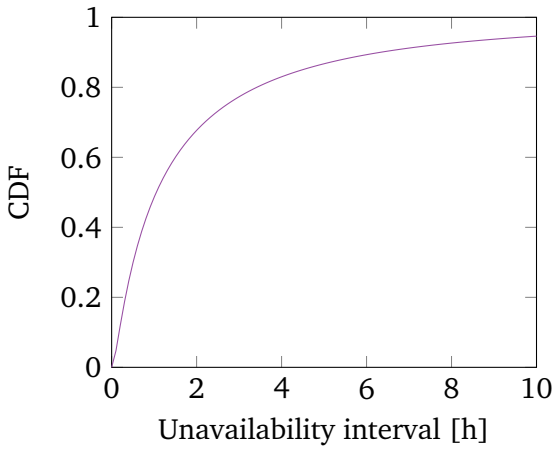


Figure 6.16: CDF for log-normal distribution of node repair time.

Table 6.4: Summary of parameters for the failure model.

Characteristic	Model	Parameters
System MTTI=1h	Weibull distribution	$\alpha = 0.7$, $\lambda = 0.8$
Node repair time	Log-normal distribution	$\mu = 0.05$, $\sigma = 1.4$

6.4 Experimental Results

For the purpose of evaluation, we simulate a cluster computer with a size of 125 nodes with 4 processors per node, giving a total of 500 processors. Three different workloads, which are listed in Table 6.5, are investigated. As mentioned earlier, the base scheduling algorithm is a FCFS scheduler with support for moldable jobs, malleable jobs and backfilling. Moldable jobs are scheduled using the SA scheduling algorithm. Malleable jobs are scheduled with the algorithms DBES, PRA and PWA in separate experiments to draw a comparison. A description of SA, PRA and PWA is available in Chapter 2. They are briefly explained below for a quick reminder.

1. **SA - Supercomputer AppLeS (SA)**
   It is a moldable job scheduling algorithm proposed by Cirne [85], which allocates nodes to jobs with the aim of delivering the best turnaround time for each job.

2. **PRA - Precedence to Running Applications**
   It is a malleable job scheduling algorithm proposed by Buisson et. al. [83] that expands all running malleable jobs whenever idle nodes are available in the system instead of using them to start new jobs or perform backfilling.

3. **PWA - Precedence to Waiting Applications**
   It is a malleable job scheduling algorithm proposed by Buisson et. al. [83] that uses idle nodes to start new applications rather than expanding running malleable jobs. The nodes that still remain idle after this step are then used to expand running malleable jobs.

Table 6.5: Composition of workloads.

Workload name	Job types		
	Rigid	Moldable	Malleable
Mixed workload	400	300	300
Moldable-rigid workload	500	500	-
Malleable-rigid workload	500	-	500

Also, the simulation assumes perfect checkpointing. That is, it is assumed that a job that is affected by a node failure can be restarted from exactly the point at which it was interrupted by the node failure.

6.4.1 Mixed workload

The mixed workload consists of 400 rigid jobs, 300 moldable jobs and 300 malleable jobs. The base scheduling algorithm used for scheduling the workload includes FIFO with conservative backfilling for rigid jobs and SA for moldable jobs. The malleable jobs were scheduled using PRA, PWA, and DBES in separate experiments. Figure 6.17 shows the makespan of the workload scheduled with the above mentioned algorithms under different failure conditions. The rigid scheduling strategy treats all jobs as rigid jobs with FIFO and backfill scheduling.

In the absence of node failures, one would expect that scheduling the workload with the awareness of malleable and moldable jobs will result in a lower makespan than scheduling all of them as rigid jobs. It can be noticed that while this is true for PWA and DBES, PRA has a larger makespan than even rigid scheduling. This is a result of prioritizing the use of idle processors for expanding malleable jobs instead of starting new ones. While expanding a malleable job can improve its speedup, it may not always deliver the best parallel efficiency. Thus the makespan in this cases increases more than even that of rigid scheduling as the execution of several malleable jobs had lower parallel efficiency. Between PWA and DBES, DBES performs negligibly better than PWA. These observations only further confirm the conclusions of Chapter 5 that DBES can consistently perform better than other malleable job scheduling algorithms.

In the presence of failures and without using dynamic node replacement, the makespan of all the scheduling strategies increases as expected because of the regular node failures. The makespan of the rigid scheduling strategy in the presence of node failures shoots up substantially by about 15 hours when compared to the makespan of the same without node failures. On the other hand, the makespan of PRA, PWA and DBES in the presence of node failures shoot up only by an average of 5 hours when compared to their own makespan without node failures. This is because of malleable and moldable jobs that can use resources flexibly and cause less wastage of resources even as failures occur. This shows that even when experiencing a high failure rate, simply enabling moldable and malleable job scheduling can bring about better system utilization and throughput than rigid scheduling.

When the dynamic node replacement algorithm is used, the makespan of adaptivity-aware scheduling strategies is further reduced to stay close to that of the scenario without node fail-

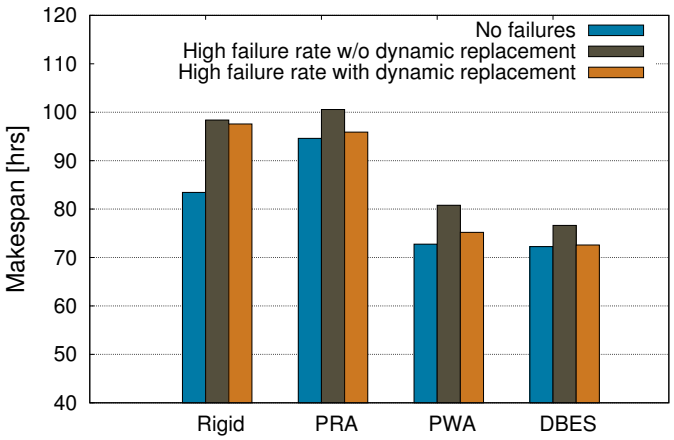


Figure 6.17: Time for completion of the mixed workload with various scheduling algorithms.

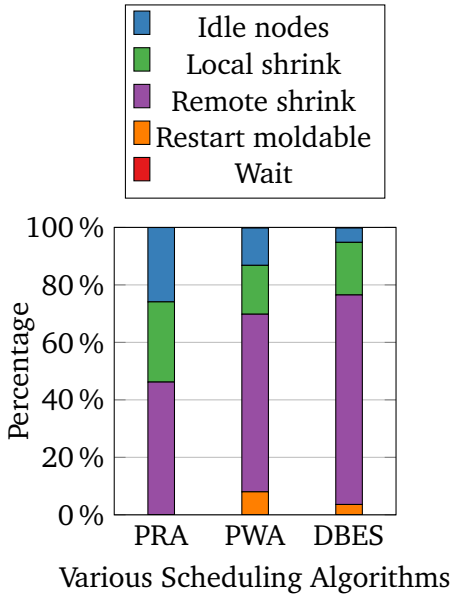


Figure 6.18: Node replacement sources in the mixed workload.

ures. Among the three scheduling algorithms, DBES delivers the smallest difference between the makespan in the case of no failures and high failures with dynamic node replacement. Therefore, the dynamic node replacement algorithm can immensely improve the scheduling performance that sees a dip due to hardware failures.

Figure 6.18 shows the break up of the percentages of the sources from which replacements were found when the dynamic node replacement algorithm was used under PRA, PWA, and DBES. We can observe that the majority of the replacement nodes were found by shrinking other running malleable jobs under all three strategies. This percentage is considerably higher than the percentage of replacement nodes found from idle ones. This is a result of enabling malleability which increases system utilization and leaves less idle nodes. The percentage of idle nodes used for replacement is highest in PRA compared to PWA and DBES because of the relatively lower system utilization maintained by PRA compared to PWA and DBES. The fact that DBES has the least percentage of idle resources used as replacements is a direct implication of the conclusions from Chapter 5 – that due importance must be given to backfilling as well when scheduling a malleable workload. While it has already been clearly established that the dynamic node replacement algorithm curbs the performance loss irrespective of the base scheduling algorithm used, its combination with the DBES produces the best performance due the similar principles of job selection used by these two algorithms. This is described in detail below in the Section 6.4.2.

Another important observation is the role of moldable jobs and the local/remote restart of moldable jobs by the dynamic node replacement algorithm. We can see from Figure 6.18 that the percentage of moldable jobs restarted to find replacements for handling a failure has been far less compared to other replacement sources. Restarting a moldable job is typically an expensive operation as the job has to start from the beginning. Therefore, the lower the number of moldable jobs restarted is, the better is the overall performance. However, this step in the

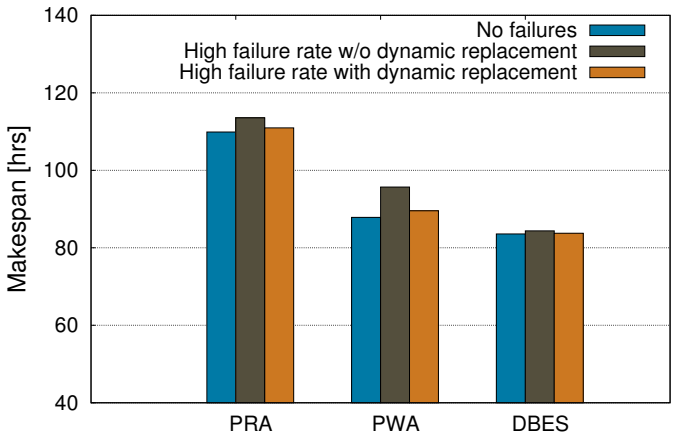


Figure 6.19: Time for completion of the malleable-rigid workload with various scheduling algorithms.

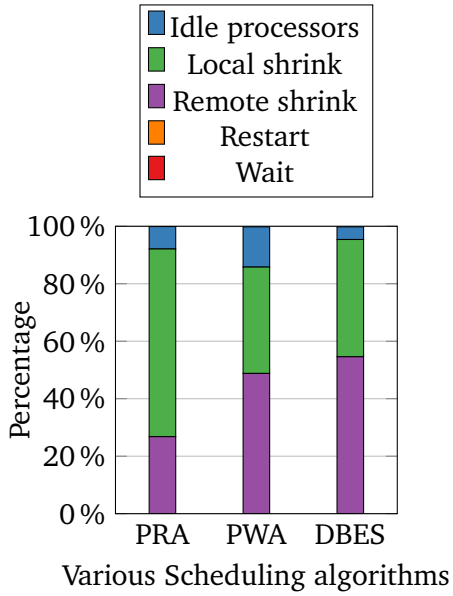


Figure 6.20: Node replacement sources in the malleable-rigid workload.

dynamic node replacement algorithm was vital in effectively reducing the number of jobs that needed to be re-queued as a result of not having found any replacement to zero in all the three strategies. For example, when the step for considering the restart of moldable jobs was disabled in the dynamic node replacement algorithm used with DBES as base, 8 jobs were re-queued out of the 220 jobs that were affected by failures. Thus, although its contribution in finding replacements is lower compared to other replacement sources, it plays an essential role and has the potential to make a larger contribution in the future as checkpoint/restart techniques mature. This is described in detail in Section 6.4.3.

6.4.2 Malleable-rigid workload

The malleable-rigid workload consists of a total of 1000 jobs with 500 malleable jobs and 500 rigid jobs. The makespans of this workload with the PRA, PWA and DBES schedulers and various failure scenarios are shown in Figure 6.19. The corresponding sources of replacement are shown in Figure 6.20. It can be observed that the results follow the same trend as the mixed workload. Without the presence of failures PRA has the longest makespan and DBES has the shortest. Similar to the mixed workload, the presence of node failures increase the makespan with all three base scheduling algorithms. When the dynamic node replacement algorithm is applied, the makespan is brought down close to the original makespan without node failures.

However, the most important observation in this workload is that the makespan with the DBES scheduling algorithm is almost the same in all three cases: absence of node failures, high node failure rate without dynamic node replacement, and high node failure rate with dynamic node replacement. This is because DBES performs an implicit dynamic node replacement. When a job is affected by node failures and no dynamic node replacement algorithm is present, the

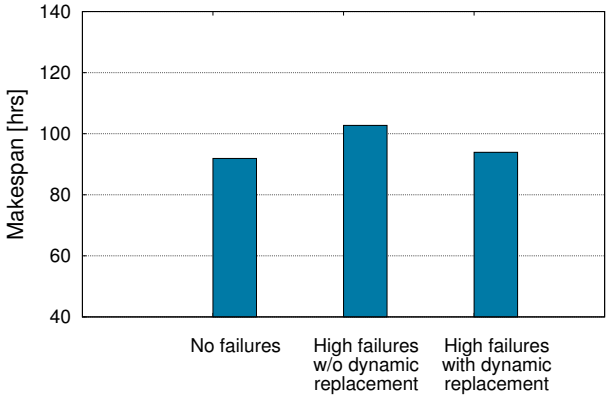


Figure 6.21: Time for completion of the moldable-rigid workload in various scenarios.

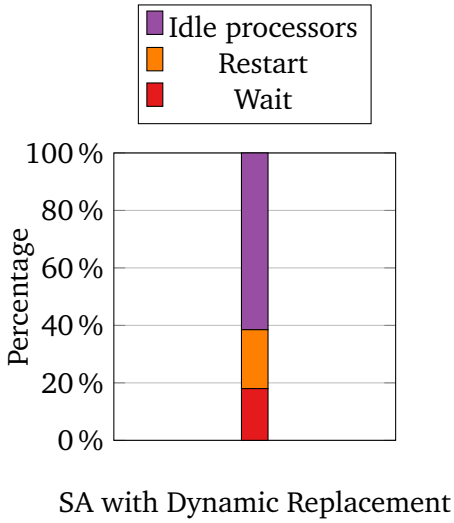


Figure 6.22: Node replacement sources in the moldable-rigid workload.

job is put back into the idle/pending queue with the highest priority. Therefore, in the next scheduling iteration, DBES can already allocate resources for this job through its basic scheduling mechanism. This is done by either providing idle healthy nodes or by obtaining them from shrinking other running malleable jobs. This is similar to the steps followed by the dynamic node replacement algorithm.

While the steps taken by the DBES and the dynamic node replacement algorithm are largely similar, they are not equivalent. The dynamic node replacement algorithm considers the type of job that is affected when making replacement decisions. Therefore, when the failed job is a malleable job, it will try to shrink the malleable job by removing the failed nodes and leave it to the base scheduling algorithm for a later expansion. On the other hand, without dynamic node replacement algorithm, any base scheduling algorithm will re-queue the job and try to restart it with the exact the number of nodes that the job held before being affected by node failure. Therefore, the makespan of the workload with high failure rate and without dynamic node replacement can grow much larger compared to a scenario without any failures depending upon the workload.

6.4.3 Moldable-rigid workload

The moldable-rigid workload consists of 500 rigid and moldable jobs each, totaling to 1000 jobs. As already described, moldable jobs are scheduled using the SA algorithm. Figure 6.21 shows the makespan of the workload in the three failure scenarios: no failures, high failure rate without dynamic node replacement, and high failure rate with dynamic node replacement.

The moldable-rigid workload also exhibits the same performance trend as the mixed and malleable-rigid workload. The makespan of the workload when subjected to a high node failure rate increases by about 10 hours as compared to having no hardware failures. However, with the dynamic node replacement strategy, the makespan is only greater by 2 hours than that of

the scenario without any failures. The split-up of the sources of replacement when using the dynamic node replacement algorithm is shown in Figure 6.22.

A high percentage (62%) of instant replacements came from idle processors due to the low system utilization (65%) obtained when scheduling this workload. However, the continuing node failures did not allow instant replacements all the time and some jobs had to wait until nodes were released by the normal termination of other running jobs. This constituted about 18% of the node replacements. Approximately 20% of instant replacements were provided by restarting moldable jobs (step 4 of the dynamic node replacement algorithm).

Thus, almost an equal percentage of replacements had been provided by restarting moldable jobs and waiting for node releases. The strategy of restarting moldable jobs alone could not completely avoid the waiting. This is typical because restarting moldable jobs is an expensive operation. Unlike malleable jobs that can seamlessly continue execution after a shrink or an expand step, moldable jobs have to be restarted from the beginning of the application. The results of the execution performed before terminating a moldable job cannot be used after restarting it with a different number of processors. Therefore, this strategy makes only a smaller contribution towards finding instant node replacements as compared to using malleable jobs or idle resources for finding replacements.

However, this strategy can be expected to play a more prominent role in the future as checkpoint/restart techniques advance. Current disk-based checkpoint/restart techniques are only able to restart a job with the same number of processes from a checkpoint. On the other hand, restarting jobs with a different number of processors from a checkpoint is one of the important research directions in checkpoint/restart techniques and is not far from being a reality. Since moldability is easier to achieve than malleability, this replacement strategy will be able to take advantage of the improved checkpoint/restart methods and the larger number of moldable jobs that may be present in future workloads.

6.5 Summary and Conclusion

Although the computational demands of user applications drive the effort to create exascale systems, high computational power at the cost of reliability is not desired by any user. Given that exascale systems are predicted to have a MTBF of one hour or less, achieving high resiliency is one of the top goals in making exascale systems real. Until now, applications could use the libraries for checkpoint/restart or replication separately for fault tolerance. However, moving forward, exascale systems call for an approach of tightly coupling multiple middleware components to achieve resiliency.

To this end, this work proposes a dynamic node replacement algorithm which replaces failed nodes of a job with healthy ones on-the-fly. The algorithm uses the unique characteristics of all job types to find fast replacement nodes. The algorithm is implemented as a supplemental algorithm to a base scheduling algorithm and is triggered in the event of node failures. The algorithm was implemented on a custom-discrete event simulator based on the GridSim simulator. The evaluation shows that dynamic node replacement has a strong potential to curb the throughput loss that inevitably occurs when experiencing high failure rates.

The dynamic node replacement algorithm is a perfect compliment to the new multi-level checkpointing features that are increasingly becoming popular. While this contribution used a simulator to demonstrate the benefits of the algorithm, it opens the scope for building a wide variety of interfaces between checkpoint/restart frameworks and the batch system to achieve functionality such as automatic checkpointing and proactive migration. Also, the batch system could use the checkpoint/restart framework to achieve pseudo-malleability in applications if they can be restarted from a checkpoint with a different number of processors. Overall, the role of batch systems in future cluster environments will be key to making systems more robust.

7 Dynamic Resource Management in Architectures with Network-Attached Accelerators

In this chapter we present the applicability of the proposed dynamic resource-management facilities to cluster architectures with network-attached accelerators. We describe the DEEP cluster system as a motivating example of an architecture designed with network-attached accelerators towards reaching the goal of exascale. Thereafter, we describe the dynamic accelerator-cluster architecture, which is a prototypical system designed to study the certain resource-management aspects of the DEEP system and present an enhanced version of TORQUE/Maui batch system that is particularly suitable for such architectures. Finally, we evaluate the batch system and present our conclusions.

7.1 The DEEP Cluster System

Today, heterogeneous architectures are realized mainly by attaching accelerators such as GPUs to every node of the cluster. A drawback in this architecture are the latency penalties and bandwidth limitations of the GPU-to-GPU communication over the PCIe bus. Moving towards exascale, many approaches are being explored to address such drawbacks in current accelerator-cluster environments.

The DEEP (Dynamical Exascale Entry Platform) project [130, 63] proposes a novel cluster-booster architecture and builds the first incarnation of the same, where all the compute nodes can access a cluster of accelerators. This enables fast accelerator-to-accelerator communication and flexible use of accelerators by compute nodes. An illustration of the cluster-booster architecture is presented in Figure 7.1.

The cluster part of the system comprises nodes connected through Infiniband. Each cluster node (CN) hosts two Intel Xeon E5 2680 processors. The booster is a cluster of accelerators connected through a torus network and built out of so-called *booster node* (BNs). Each BN hosts an Intel Xeon Phi 7120X coprocessor [1] (developed under code name Knights Corner or KNC) as the accelerator, which consists of 61 compute cores running an enhanced version of the x86 instruction set. The KNC is designed as a card connected to a host processor by PCIe and is capable of autonomously booting, running its own operating system, and performing network communications. However, in the DEEP system, it is used as a standalone component without using a host processor by connecting the KNCs through the novel EXTOLL interconnect [4]. The Booster Interface (BI) connects the network of the cluster with the network of the booster. A global MPI environment extending across the cluster and booster part is provided as the programming environment of the DEEP system. Scalable part of an application are executed on

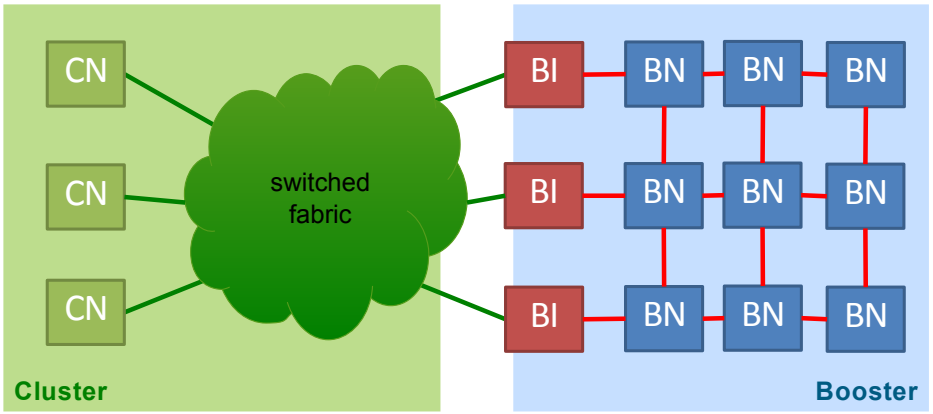


Figure 7.1: The DEEP cluster-booster architecture [1].

the booster while the less scalable part are executed in the cluster. Computations are spawned on the the BNs using the MPI-2 process management facilities [131]. The OmpSS programming model is provided for application developers, which uses the underlying MPI. Thus, existing MPI and OmpSs applications can easily use the DEEP system with only moderate modifications.

DEEP not only enables fast accelerator-to-accelerator communication, but also flexible use of accelerators by compute nodes. It allows both static and dynamic allocation of the accelerators to compute nodes based on application request — similar to evolving jobs. The dynamic resource-management facilities developed in this thesis are deployed in the DEEP system by integrating them with the ParaStation Cluster Suite [132].

7.1.1 Dynamic booster node allocation in the DEEP cluster system

Figure 7.2 presents the workflow of the TORQUE/Maui batch system and the ParaStation Cluster Suite in the DEEP system. Unlike in other cluster environments, the `mom` of the TORQUE resource manager is replaced by the `psid` control daemon from the ParaStation Cluster Suite. `psid` provides functionality for the process management of the ParaStation MPI. It consists of a ParaStation MOM (`psmom`), implemented as a plugin, which provides all the features of TORQUE's `mom`. The ParaStation MOM offers fast and reliable job startup along with unique features such as:

- Advanced accounting for all processes of serial and parallel jobs
- Secure communication between the ParaStation process management and TORQUE
- Reliable and highly scalable node-to-node communication
- On-the-fly reconfiguration and information gathering

Dynamic resource management is realized by using the enhanced TORQUE/Maui batch system with an extended `psmom` and ParaStation PMI (Process Management Interface). Unlike the approach for evolving jobs presented in Chapter 4, dynamic allocation of BNs is established transparently when offloading computations on the BNs.

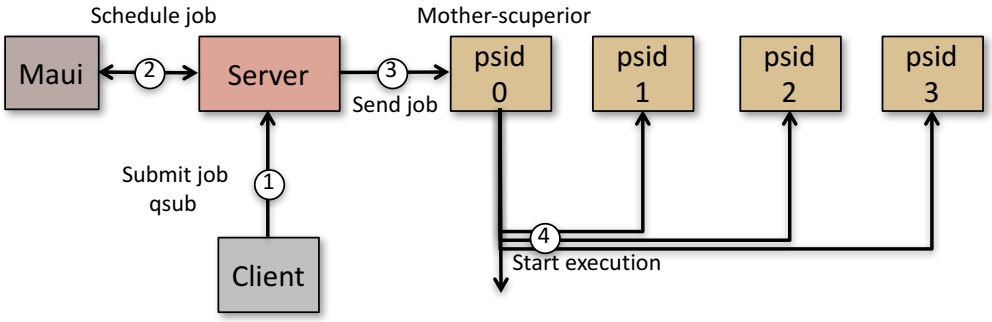


Figure 7.2: Workflow of the TORQUE/Maui batch system in combination with ParaStation Cluster Suite.

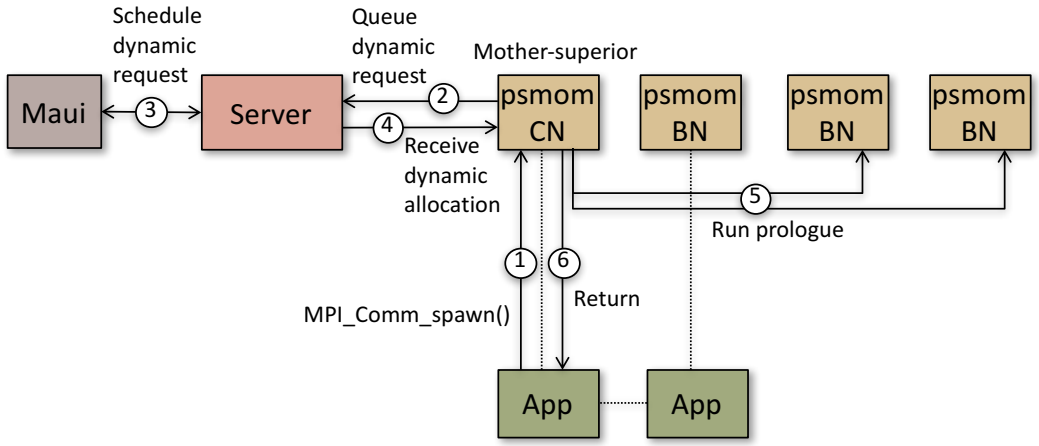


Figure 7.3: Workflow of dynamic allocation in DEEP.

Listing 7.1 shows the use of `MPI_Comm_spawn()` to offload computations onto the BNs and Figure 7.3 shows an example of the corresponding actions taken by the batch system components thereafter in the DEEP system. The example scenario considers one CN with one statically allocated BN and two dynamically allocated BNs. An `MPI_Comm_spawn()` message is sent to the local `psid` which is forwarded to the mother-superior `psid`. If there are not sufficient BNs available to the spawn from the BNs that were statically allocated, the `psmom` plugin initiates a dynamic request for the missing resources. When the resources are allocated by Maui, the TORQUE `server` forwards the information to the mother-superior `psid`. The mother-superior `psid` starts the prologue on the dynamically allocated nodes and spawns the processes appropriately. Once a spawned process finishes execution, the resources used by this process will be freed so that they can be used by other jobs. This occurs when the processes executing on the dynamically allocated BNs call `MPI_Finalize()` and subsequently terminate.

Thus, the dynamic resource-management facilities, particularly when integrated with MPI, allow the convenient management of network-attached accelerator resources. These features can be useful for accelerator-equipped systems and architectures in upcoming exascale systems.

Listing 7.1: Computational offloading in DEEP.

```
void main(int argc, char **argv) {

 /* Init the MPI */
 MPI_Init(&argc, &argv);
 ...
 /* Load computations on BN */
 MPI_Info      info;
 MPI_Info_create(&info);
 MPI_Info_set(info, "nodetype", "booster");
 MPI_Comm_spawn("child",
       MPI_ARGV_NULL,
       2,
       info,
       0,
       MPI_COMM_WORLD,
       &intercomm,
       MPI_ERRCODES_IGNORE);
 ...
 /* Finalize with the spawned processes */
 MPI_Finalize(intercomm);
 ...
 /* Finalize MPI */
 MPI_Finalize(MPI_COMM_WORLD);
}
```

Since the DEEP system is still under development with respect to hardware and network interconnect, the evaluation of the dynamic resource management techniques for such architectures is shown with a similar architecture called the *dynamic accelerator-cluster architecture* in the next section.

7.2 Dynamic Accelerator-Cluster Architecture

The dynamic accelerator-cluster (DAC) architecture was proposed by Rinke et. al. [2] as a prototype of the DEEP architecture in order to evaluate aspects such as computational offloading on remote accelerators and flexible resource management. Similar to the DEEP architecture, accelerators are loosely coupled to compute nodes. That is, instead of being directly connected to a compute node (e.g., by PCI Express), an accelerator is attached to a high-speed interconnection network (e.g., Infiniband) which is shared among all compute nodes and accelerators (see Figure 7.4).

Loosely coupling accelerators and compute nodes prevents compute nodes from selecting and using accelerators right away. Instead, accelerators must be mapped to a compute node before the compute node can offload computations onto them. This dynamic mapping is performed by the accelerator resource manager (ARM). The ARM assigns unused accelerators to compute nodes based on application demands communicated by the compute nodes. Once accelerators are assigned to a compute node, the application running on this compute node can

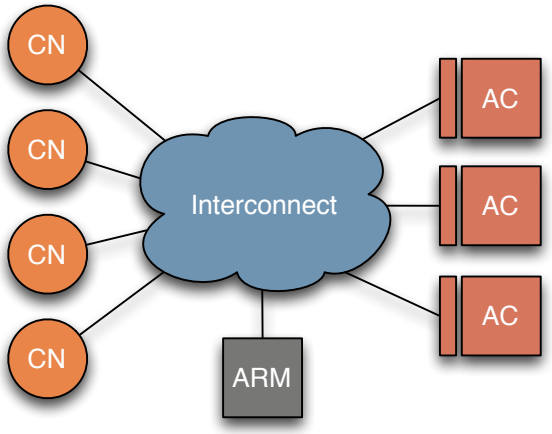


Figure 7.4: The dynamic accelerator-cluster architecture (CN - compute node, AC - accelerator, ARM - accelerator resource manager).

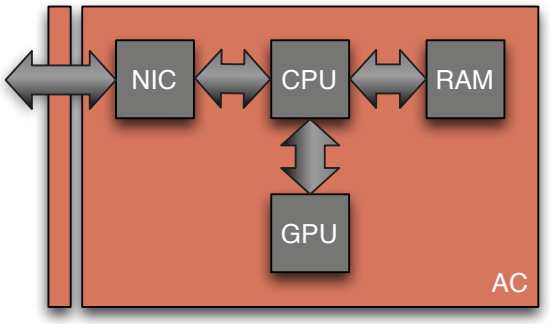
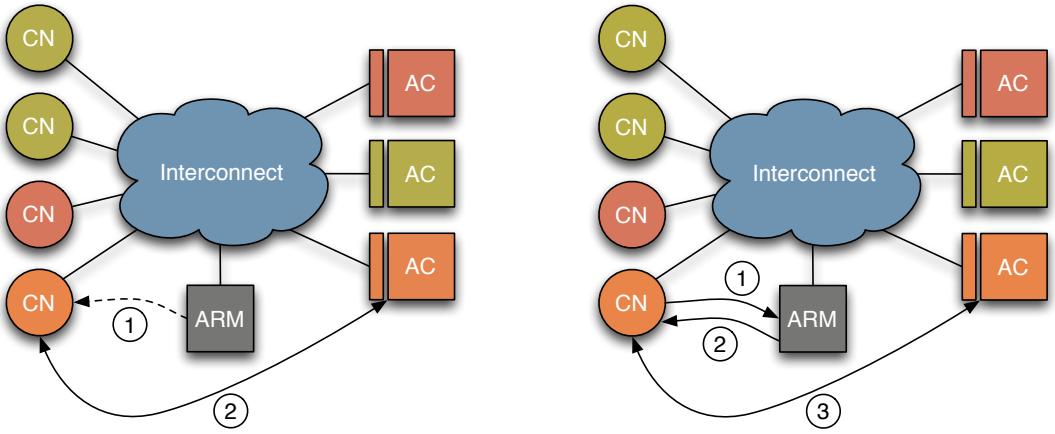


Figure 7.5: Accelerator in the dynamic accelerator-cluster architecture.

directly access those accelerators. While in the DAC architecture the compute node and the cluster interconnect are similar to that of the popular cluster architecture, *accelerator* is defined differently.

Accelerators. There exists a fundamental difference between accelerators as described in popular terminology (like GPUs) and the accelerators as described by this architecture. An accelerator in the context of the DAC architecture is actually a node consisting of one or more accelerators (e.g., GPUs) rather than the accelerator (e.g., GPU) working as a standalone entity that is directly connected to the network. The basic structure of such an accelerator is illustrated in Figure 7.5. The accelerator comprises an energy-efficient CPU with its main memory (RAM), a network adapter (NIC), and one or more accelerators of the type that is currently available on the market to accelerate computations (e.g., a GPU as depicted in the Figure 7.5). The CPU runs an operating system which is able to instruct the network adapter to initiate network transfers to other devices. Therefore, such an accelerator is able to communicate with compute nodes as well as with other accelerators over the interconnection network. Data is transferred over the network interconnect to the main memory of the accelerator before being transferred to the GPU.



(a) Static assignment.

(b) Dynamic assignment.

Figure 7.6: Static (a) and dynamic (b) accelerator assignment. Different shadings denote different jobs. Dashed lines denote communication before job start, whereas solid lines denote communication at runtime [2].

7.2.1 Execution model and accelerator assignment strategies

The execution model of this architecture consists of three steps: (i) accelerator allocation, (ii) accelerator usage and (iii) accelerator deallocation. Accelerator allocation can be carried out in two distinct assignment strategies. In the *static assignment strategy*, the required number of accelerators is allocated to compute nodes before job start. These accelerators remain allocated to the compute nodes until job termination. The compute nodes are provided with a computation-offload API similar to CUDA and OpenCL to offload work onto accelerators. In order to identify accelerators, a unique *handle* for each allocated accelerator is used with the API. In the *dynamic assignment strategy*, the accelerators are (de)allocated at job runtime. Compute nodes send a runtime request to the ARM to acquire new accelerators. Note that it is not guaranteed that a dynamically requested resource will always be available for the application. Subject to availability, the ARM allocates the resources to the requesting compute node. When the requested number of accelerators is not available, the ARM rejects the request. Therefore, users also take into account that the dynamic requests may not always be successful. When a dynamic request is rejected, the application continues its execution with the existing allocated accelerators. When the dynamically allocated accelerators are not needed anymore, the compute nodes can release the accelerators. For dynamic (de)allocation the compute nodes use a resource management API which complements the computation-offload API. A prototypical implementation of the ARM was developed to enable the static and dynamic allocation strategies. Figure 7.6 illustrates the static and the dynamic assignment scenarios as explained above. Dashed lines between the ARM and the compute nodes represent communication before job start as in the case of a static assignment. Solid lines between the ARM and the compute nodes represent communication at runtime depicting a dynamic assignment.

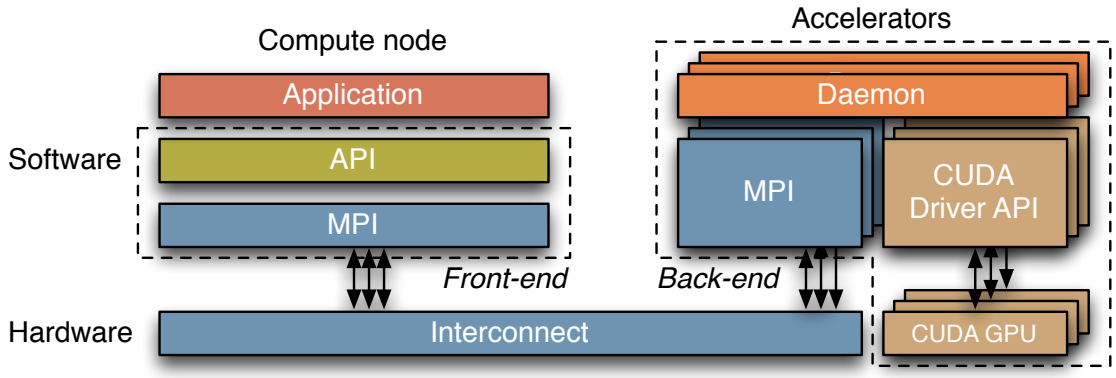


Figure 7.7: The software stack of the dynamic accelerator-cluster architecture.

7.2.2 Prototype

As stated earlier, the current version of the DAC Architecture enables computation offloading to the network-attached accelerators consisting of a CUDA-enabled GPU. Computations are CUDA kernels which are executed on the remote GPU.

The computation-offload API provides functionality to (i) allocate memory on an accelerator, (ii) copy data to or from an accelerator and (iii) launch compute kernels on an accelerator. Figure 7.7 illustrates the software stack of the DAC Architecture. It consists of a front-end on every compute node and a back-end on every accelerator. The front-end translates API calls into requests which are redirected to the back-end, where a daemon receives those requests and executes them on the CUDA-enabled GPU using the CUDA driver API. The front-end uniquely identifies the back-end through a *handle* and enables transparent communication between the compute node and the accelerator. Listing 7.2 illustrates the usage of both the computation and the resource management API with regards to using a remote CUDA-enabled GPU. Here, similar to CUDA, a computation kernel is executed on the accelerator after allocating memory and transferring data to the device. After the kernel execution is complete, data is transferred back and the memory is freed. The *ac_handle* uniquely identifies the accelerator on which the operations are to be performed. The actual communication between the compute nodes and the accelerators is accomplished through a distinct communication protocol based on MPI. Clearly, for the compute nodes to be able to communicate to the daemons running on the accelerator nodes through MPI, they have to reside in the same MPI communicator. A resource management library, which makes use of MPI-2 dynamic process management facilities, is provided to the compute nodes to establish this transparently with the accelerators.

In the resource management API, the `AC_Init()` initializes the accelerator usage for the computation-offload API after creating an MPI communicator with the statically allocated accelerators and providing a valid *ac_handle*. The `AC_Get()` call is used to dynamically request additional accelerators from the ARM. Users may use the `AC_Free()` call to release dynamically assigned accelerators. The `AC_Finalize()` routine must be called at the end and releases all the associated accelerators.

In enabling batch system support for the DAC Architecture, the functionality of the ARM is completely integrated into the batch system. The resource management library communicates directly with the batch system. Also, note that although using remote GPUs involves additional communication overhead through the cluster interconnect, applications consisting of kernels with high arithmetic intensity can still benefit from multiple accelerators. In particular, multiple accelerators can also be used with latency hiding techniques to reduce the visible communication overhead. Furthermore, the implementation also provides an efficient communication protocol which includes pipelining large data transfers, thereby optimizing the overall data transfer [2].

Listing 7.2: Example program on the Dynamic-Accelerator Cluster Architecture.

```
void main(int argc, char **argv) {

 /* Init the accelerators */
 AC_Init(&ac_handle);
 ...
 /* Allocate memory on device */
 acMemAlloc(cudaMalloc_args,ac_handle);

 /* Transfer memory to device */
 acMemCpy(cudaMemcpy_args,ac_handle);

 /* Execute kernel */
 acKernelCreate(k_name,ac_handle);
 acKernelSetArgs(k_args);
 acKernelRun(k_name,dimGrid,dimBlock);

 /* Transfer memory to host */
 acMemCpy(cudaMemcpy_args,ac_handle);

 /* Free memory on device */
 acMemFree(cudaFree_args,ac_handle);
 ...
 /* Get more accelerators */
 AC_Get(count, &ac_handle_new);
 ...
 /* Free the dynamically obtained accelerators */
 AC_Free(&ac_handle_new);

 /* Finalize */
 AC_Finalize(&ac_handle);
}
```

7.2.3 Reviewing the dynamic accelerator-cluster architecture

The dynamic accelerator-cluster architecture addresses the disadvantages of the current static mapping of accelerators to compute nodes. In general, the architecture provides the following advantages.

1. **Flexibility before runtime (better matching of accelerators to compute nodes)**
   While in the popular cluster architecture, the applications are constrained either by the number of accelerators physically attached to a compute node or the type of accelerator attached to the compute node, by enabling a loose coupling, this architecture extends the freedom of applications in terms of using accelerators available throughout the cluster system.

2. **Flexibility in using accelerators at runtime**
   The dynamic mapping adds useful flexibility in using the available accelerators. By supporting the dynamic assignment strategy, the architecture allows the number of accelerators an individual application is using to be varied during the lifetime of the job. This enables users to use accelerators according to the needs of different computation phases in an application.

3. **Fault Tolerance**
   The loose coupling avoids an implicit correlation between compute nodes and accelerators. Hence, broken accelerators or compute nodes do not affect each other as in the case of cluster systems using a static mapping of accelerators to compute nodes, where, for instance, broken compute nodes could prevent PCI Express-connected accelerators from being used.

4. **Increased hardware utilization**
   Since accelerators are tightly coupled to compute nodes in traditional architectures, they remain unused when applications that only use CPUs are executed on a compute node. Since the loose coupling eliminates such a restriction, the unused accelerators can be assigned to any compute node increasing the overall hardware utilization of accelerators in the cluster environment. Moreover, exclusive access to accelerators can also be guaranteed.

5. **Cost Efficiency**
   In the dynamic accelerator-cluster architecture, the number of compute nodes and accelerators can scale independently. That is, due to loose coupling, accelerators and compute nodes can be added to the cluster independently of each other.

While the additional network transfer may incur some overhead in terms of application performance, it is minimized by the use of faster interconnects such as Infiniband. Since the software stack uses MPI for network communications, support for all future high-speed network interconnects is automatically guaranteed and the overhead is minimized. Applications may

also experience additional overhead when using the ARM to request accelerators. Therefore, performance considerations are key factors when designing the ARM.

7.3 Dynamic Resource Management and Scheduling for the DAC Architecture

In general, the dynamic resource management and scheduling facilities presented for evolving jobs in Section 4.1 were employed to establish both static and dynamic accelerator assignment strategies. While the scheduling facilities in Maui were used unchanged, the resource management techniques in TORQUE were slightly enhanced with the awareness of network-attached accelerators and the DAC execution environment. For the static assignment, before executing the application on the compute nodes, TORQUE ensures that appropriate daemons are started on the accelerators in order to be used by the compute nodes. We have extended the `server` and the `moms` to perform the above in the DAC environment. For the dynamic assignment, applications were made to communicate faster with the `server` directly through the PBS IFL API instead of the TM API unlike the approach presented earlier in Section 4.1. The resource management library uses the `pbs_dynget()` call to request additional accelerators. Similarly, to dynamically release accelerators, the `pbs_dynfree()` call has been added to the Interface Library and the `moms` have been extended with capabilities to dynamically disassociate themselves from the job. The reason for choosing direct communication between the job and the `server` in the DAC architecture is its conceptual design. Since each compute node may use an independent set of accelerators with unique handles, its dynamic requests are also exclusive of other compute nodes. Therefore, enabling direct communication with the `server` in this case is faster and avoids unnecessary overheads of sending the requests first to the `moms`. The operation of the batch system for static and dynamic accelerator assignment is described below.

7.3.1 Static allocation of network-attached accelerators

In the static allocation scenario, a job requests a particular number of network-attached accelerators during job submission time. The job is not executed until all the required resources are available. During job start, the resource management library establishes the association with the accelerators. Users may then use the compute node API to offload computations to the accelerators. We consider the example of one compute node requesting x accelerators and describe the operations of both the batch system and the resource management library, separately when assigning the accelerators to the compute node. Figure 7.8 illustrates the scenario with one compute node statically associating itself with three accelerators.

Batch system. The user requests a DAC execution environment using the extended `qsub` command as shown below.

```
$ qsub -l nodes=1:acpn=x jobscript.sh
```

The `acpn` job attribute indicates the request of x network-attached accelerators per compute node. For a multi-compute node job with k compute nodes, this request would mean a total

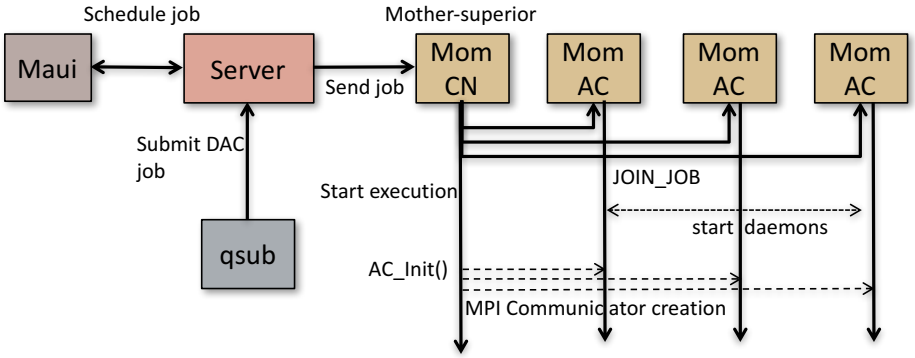


Figure 7.8: Workflow of a static allocation in the DAC architecture.

of k compute nodes and $k \times x$ accelerators for the job. The `server` then enqueues the job for scheduling. The Maui scheduler allocates the required resources and informs the `server`. The `server` selects the `mother-superior` (which is always a compute node) and forwards the job information. Once the `moms` joins with each other, the `mother-superior` invokes the execution of accelerator daemons the accelerator nodes. The daemons are started such that each set of the accelerators to be associated with a compute node is contained under a single `MPI_COMM_WORLD`. The user application is then started on the compute nodes and the resource management library establishes the connections with the daemons.

Resource management library. As stated earlier, the resource management library uses MPI-2 dynamic process management facilities to enable a persistent connection between the compute nodes and the accelerators. Once the daemons are started, the `root` (MPI rank 0) of the accelerator daemons opens an MPI port (using `MPI_Open_port()`). The port information is made available to the compute nodes through a file. When the `AC_Init()` call is invoked, the compute nodes retrieve the port information and establish the connection through `MPI_Comm_connect()/MPI_Comm_accept()`. The inter-communicator returned through this operation is used to create an intra-communicator through the `MPI_Intercomm_merge()`. In the new MPI intra-communicator, the compute node holds rank 0 while all the other accelerators have a unique rank ranging from 1 to x. The handle of each accelerator consists of its unique rank in this communicator and is further used by the computation-offload library to transfer data and execute kernels. While the above scenario exemplifies a job with a single compute node, the process is essentially the same for a multi-compute node job. Each compute node would be associated to x accelerators with a distinct MPI communicator from the other compute nodes. In other words, one compute node cannot access the accelerators associated to the other compute nodes. When the job terminates, all the resources used by the job are released and made available to other jobs.

7.3.2 Dynamic allocation of network-attached accelerators

In the dynamic allocation scenario, compute nodes send a runtime request to the `server` for a finite number of additional accelerators through the `AC_Get()` routine. Upon allocation, the accelerators are associated with the requesting job and made available for use by the compute

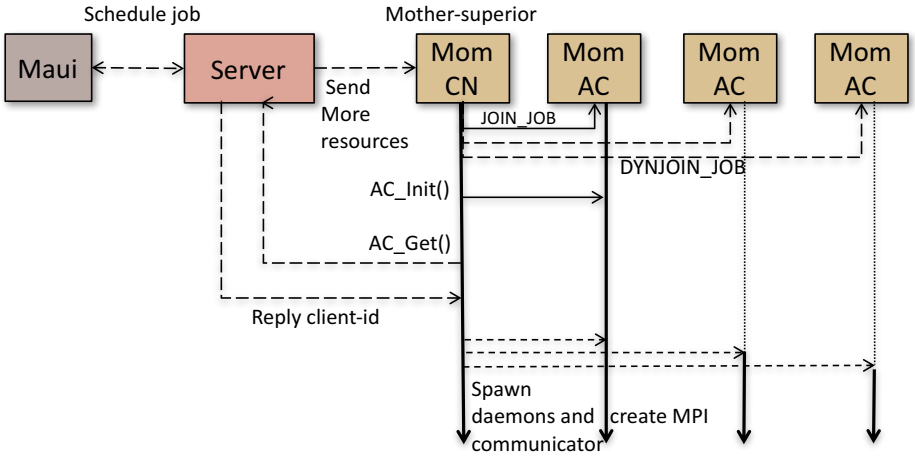


Figure 7.9: Workflow of a dynamic allocation in the DAC architecture.

nodes. As explained for the static case, we consider an example of an application with one compute node and x statically allocated accelerators, which dynamically requests y additional accelerators. Figure 7.9 illustrates the dynamic allocation scenario of a compute node with one statically allocated accelerator, requesting two additional accelerators. Dashed lines represent communication after the `AC_Get()` call and solid lines indicate communication before the call during static allocation.

Batch system. The resource management library sends the request for y additional accelerators through the `pbs_dynget()` routine which blocks until a response has been received from the server. Upon receiving the request, the `server` enqueues the job again with a special *dynamically queued* state and the Maui scheduler allocates resources for the *dynamically queued* jobs with top priority. The `server` is then informed of the allocated resources, which then forwards the information to the `mother-superior` of the job that requested the additional accelerators. Once the information has been successfully forwarded, the `server` responds to the compute node with a *client-id* which uniquely identifies this request and its set of dynamically allocated accelerators. The `mother-superior` then sends a `DYNJOIN_JOB` message to the newly allocated accelerators and also updates the existing `moms` with the addition of resources for this job. Preparing the accelerators with the daemons and establishing connection with them is performed by the resource management library. When not enough resources could be allocated for the job, the `server` rejects the request immediately with a negative valued reply. In this case, the application continues to execute with the already allocated accelerators, as stated earlier.

Resource management library. Once the additional accelerators have been allocated, the resource management library spawns the accelerator daemons on the allocated nodes through the `MPI_Comm_spawn()` call. This call returns an MPI inter-communicator with the accelerator daemons, once they have executed `MPI_Init()`. The compute node, its existing accelerators, and the newly active accelerators participate in the `MPI_Intercomm_merge()` call which results in a new intra-communicator with the compute node and all of its associated accelerators. In this intra-communicator, the compute node still holds the MPI rank 0 and the

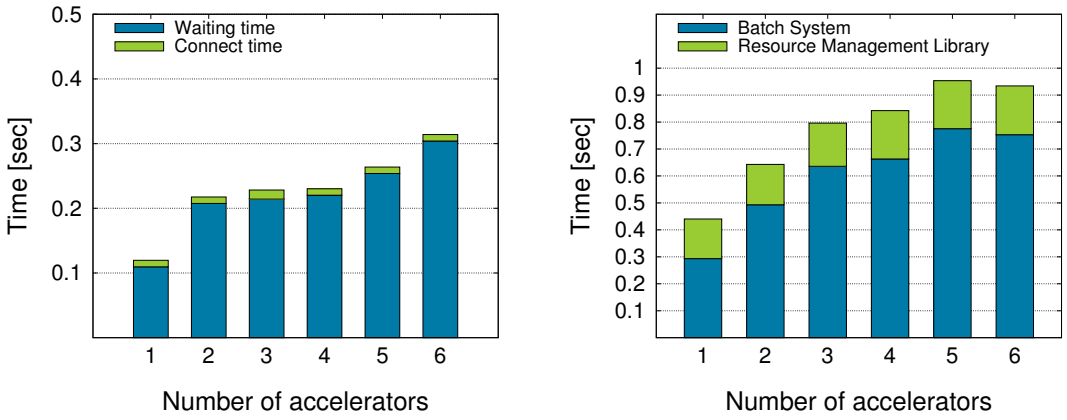
old accelerators hold their old MPI ranks ranging from 1 to x. The newly added accelerators are assigned MPI ranks ranging from $x + 1$ to $x + y$. Updated handles to the statically assigned accelerators and the new handles to the dynamically assigned accelerators are then returned to the user. These can then be used by the computation-offload library. We use `MPI_Comm_spawn()` instead of starting the daemons through the `moms` as it enables an easier way of creating the MPI communicators as opposed to using MPI Ports and employing `MPI_Comm_connect()/MPI_Comm_accept()` which is unavoidable in the case of static assignment.

Because MPI is used, a set of dynamically allocated accelerators are started with the accelerator daemons that are encompassed in one `MPI_COMM_WORLD`. Therefore, when the dynamically allocated accelerators are released, they are released as a set identified by the *client-id* through the `AC_Free()` call. The compute nodes first disconnect from the to-be-released accelerators through `MPI_Comm_disconnect()` and send the `server` the *client-id* of the set of dynamically allocated accelerators using the `pbs_dynfree()` call. The server returns a positive reply to the compute node without the need to enqueue the job again and initiates the process of disassociating the accelerator nodes from the job while the user application may continue to execute. To release the accelerators, the `server` instructs the `mother-superior` with the list of hosts that are to be disassociated from the job. The `mother-superior` sends a `DISJOIN_JOB` message to the `moms` operating on these hosts resulting in complete disassociation of these `moms` from the job. The `moms` kill all the tasks running on their host and release their resources so that they can be used by other jobs. The `mother-superior` also sends the information to the other `moms` associated with the job so as to update their database.

In the case of a multi-compute node job, each compute node may use its own `AC_Get()` to obtain additional accelerators. However, the `server`, is able to service only one request at a time per job. This may lead to long waiting time during the application runtime for some compute nodes of the job until their additional accelerators are allocated. This can be avoided using `AC_Get()` collectively over all the compute nodes that request additional accelerators. When requested collectively, one compute node collects the information about the number of accelerators required by each compute node participating in the collective call, and sends a single request to the `server` requesting the total number of required accelerators. However, the disadvantage is that either all the compute nodes get their accelerators allocated or none, since the batch system tries to allocate the total number of accelerators requested. Also, since they all obtain the same *client-id* from the `server`, they may be released only collectively. Applications can use individual or collective modes of obtaining accelerators according to their needs.

7.4 Experimental Evaluation

In this section, we present a quantitative description of the performance of our dynamic batch system in enabling static and dynamic allocation of network-attached accelerators to compute nodes in the DAC environment. Due to the novelty of this usage scenario, real world applications that use network-attached accelerators are still under development in various projects (e.g., the DEEP Project). In our evaluations, we examine the overhead of the dynamic resource allocation



(a) Time for completion of AC_Init()

(b) Time for completion of dynamic request

Figure 7.10: Time for completion of static and dynamic requests for various number of accelerators.

under various circumstances in a cluster environment with sample programs and discuss its impact on real world applications, as compared to other works that mainly simulated dynamic allocations.

For all our experiments, we used 8 nodes with 2 Intel X5570 processors at 2.93 GHz and with 24 GiB RAM each. All the nodes ran GNU/Linux 2.6.35 (Ubuntu 10.10). As MPI implementation, we used Open MPI 1.6.2. Our experiments to evaluate the batch system's performance did not require the physical presence of an accelerator in these nodes. Out of the 8 nodes, one node was designated as the `server`. It ran the `pbs_server` daemon and the Maui scheduler daemon. The same node was used as the front end. The rest of the 7 nodes were used as both compute nodes and network-attached accelerators in different test scenarios but never at the same time. All the results are an average over 10 trials.

In principle, when submitting a job, if all the required resources are readily available, the time taken to obtain the required nodes and start execution depends only on the rate at which (i) the `server` processes the request, (ii) the resources are allocated by the Maui scheduler and (iii) the `moms` receive the job information and join each other. All of the above involve communication over the network between the batch system components. If not all of the required resources are available, the requests will stay queued at the `server` until these resources become available. Since in the static allocation scenario the required number of accelerators is known prior to job start, the `server` does not start the job until all the required resources are available. Thus, the static allocation scenario is similar to the traditional way of job submission, and therefore is affected by the same parameters as mentioned above. However, to complete the static allocation, the application needs to call `AC_Init()`. Depending upon the number of accelerators requested, the `AC_Init()` call waits until all the accelerator daemons that were started on the accelerator nodes are ready to be connected to the compute node through the `MPI_Comm_connect()/MPI_Comm_accept()` routines. Figure 7.10(a) shows the time for completion of `AC_Init()` for various numbers of statically allocated accelerators ranging

from 1 to 6. The blue region depicts the amount of time spent by the call waiting until all the accelerator daemons were prepared in the remote nodes and were ready to establish a connection with the compute node. The green region depicts the time consumed in establishing the MPI communicator with the compute node. We can observe that the waiting time dominates the total time taken and generally increases with an increasing number of accelerators. However, statically allocating as many as 6 accelerators requires only around 0.3 seconds.

On the other hand, the dynamic allocation scenario introduces longer waiting time since it includes the time taken by the batch system to allocate additional resources for the job and involves the `moms` to join with each other before the accelerator daemon can be spawned on the host. Figure 7.10(b) shows the time it takes a compute node to dynamically obtain between one and six accelerators. The blue region graph indicates the waiting time, while the green region represent the time spent in performing the MPI operations, that is, spawning and creating MPI communicators. Naturally, the dynamic allocation of accelerators by the batch system dominates the overall time taken and increases with an increasing number of accelerators. The time spent performing the MPI operations is more or less the same in all the cases. While the time taken for dynamic allocation is much larger compared to `AC_Init()` in a static allocation, it still ranges only in sub-seconds for a compute node to dynamically obtain as many as six accelerators. However, the test was made under an ideal scenario where the scheduler and the resource manager are not working on scheduling jobs from a workload.

To test the behavior of dynamic allocation in the presence of other workload, we combined the scenario of a dynamic request for an accelerator along with a large number of other `qsub` requests. Since the Maui scheduler always treats dynamic requests from the DAC environment with top priority, when a dynamic request and other `qsub` requests arrive in parallel, the dynamic request will primarily be granted the resources. Therefore, the time taken for the dynamic allocation in such a scenario is similar to the time taken in the absence of any other workload. The workload may affect the performance of a dynamic allocation request only when the dynamic request arrives at the `server` precisely when the scheduler is already working on allocating resources for earlier requests. This causes additional waiting time to the dynamic allocation. In Figure 7.11 shows the time taken for the completion of dynamic requests as they reach the batch system precisely when the scheduler is already working on allocating resources for 0 (A), 16 (B) and 20 (C) other `qsub` requests. For this case we also took care that none of the 16 or 20 jobs interfered with the compute node or the accelerator node running under the DAC environment. Clearly, the larger the workload handled by Maui at the time of arrival of the dynamic request, the longer the waiting time for the dynamic request to be serviced.

Finally, Figure 7.12 compares the time taken for the dynamic allocation of one accelerator (excluding the time consumed by the MPI operations) in the case of three compute nodes (*A*, *B* and *C*) belonging to three distinct jobs sending a dynamic request each during the same time. Clearly, due the serial processing of the dynamic requests by the `server`, compute node *C*, as shown in the graph, suffered a longer waiting time.

In general, we can observe in all the scenarios that the time taken for the dynamic allocation of accelerators to compute nodes usually lies in the range of sub-seconds. For real-world

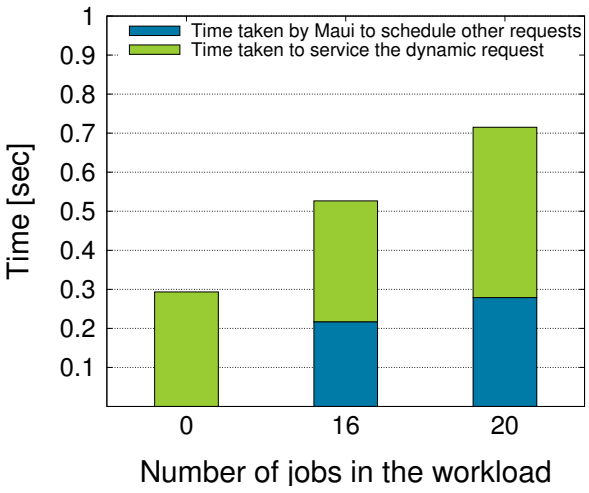


Figure 7.11: Time taken by the batch system to dynamically allocate one accelerator under different load conditions.

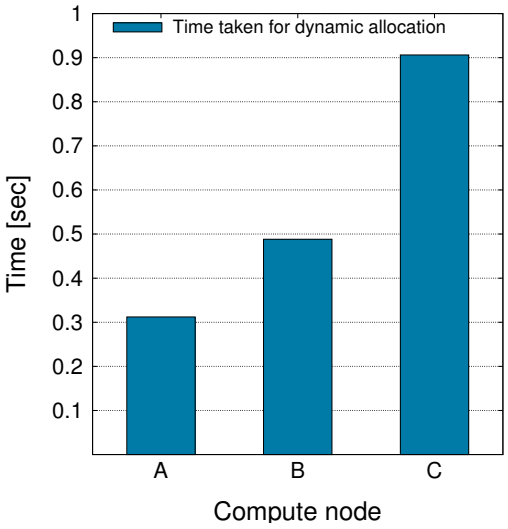


Figure 7.12: Time taken for completion of consecutive dynamic requests from three distinct compute nodes.

applications, such an overhead is negligible and may be traded off for the availability of more resources to offload computations.

7.5 Summary and Conclusion

During the last couple of years, accelerators have gained increased importance and already play a vital role in many of today's cluster systems. Given the recent advances in accelerator technology, network-attached accelerators seem to be one of the next logical steps, particularly towards reaching exascale. As network technologies progress, the latency in such architectures can drastically improve, thereby reducing the cost of frequent computational and data offloading. For example, Intel Omni-Path [133] and Infiniband EDR [134] can already provide a bandwidth of 100 Gbit/s. The roadmap of Infiniband targets reaching 600 Gbit/s by 2017 [134].

In this chapter, we presented how the dynamic resource-management facilities proposed in this thesis can be used in cluster architectures with network-attached accelerators. We presented its usage scenarios for both the DEEP cluster system and the Dynamic Accelerator-Cluster Architecture. By integrating the dynamic resource-management facilities with the programming environment, dynamic (de)allocation can be achieved transparently. This contributes to the optimized utilization of resources in a convenient way. Experimental evaluation in the DAC architecture clearly reaffirms the negligible overhead of the dynamic allocation technique.

The community is continuously striving to optimize the use of HPC hardware. Many frameworks similar to the DAC architecture, but with unique flavors, have been proposed for computational offloading on remote GPUs [135, 136, 137]. The TORQUE/Maui batch system extended for the DAC and DEEP architectures can be used with only minor modifications in all such frameworks. Thus, it is a global solution that is easily extensible to suit frameworks and architectures involving network-attached accelerators.

In a similar trend, advancements in storage technology opens the opportunity to realize network-attached storage that is different from today's parallel file system. For example, non-volatile memory such as an SSD can be attached to a subset of nodes in a cluster. These nodes can then be used as storage nodes as opposed to pure compute nodes. Jobs running on the compute nodes can store large amounts of data on the SSDs of these storage nodes, which are exclusively and dynamically allocated to them. In such an approach, applications can avoid accessing a parallel file system for every I/O operation and achieve higher I/O bandwidth. This I/O bandwidth can only be expected to improve in the future with enhancements to the way storage devices are connected in a node. For example, SSD storage over PCIe is already available from SanDisk [138] and Intel has announced the future release a non-volatile memory on DIMM form factor based on the Intel 3D XPoint Technology [139, 140]. The integration of these new devices and the new usage scenarios they offer can only be realized with support of the batch system.

8 Conclusion and Outlook

This chapter provides the summary and conclusion of the contributions of this thesis. This is followed by an outlook on the areas for future research based on this work.

The main subject of this thesis has been the development of dynamic job scheduling and resource management techniques for HPC. While there have been several advancements in static scheduling for HPC, dynamic scheduling has not seen a comparable progress from the time it was concieved. This is mainly because dynamic scheduling has never been an imperative requirement in the past. However, with the recent progresses that HPC has gone through, the motivation to investigate and propose practical methods for the same comes mainly from: (i) scientific applications exploring new domains with potentially varying resource requirements during the different phases of the application, (ii) the growing ease of realizing malleability in upcoming cluster systems, and (iii) the exploration of heterogenous architectures with various network-attached resources including accelerators and storage. Therefore, the main goal of this thesis has been to provide novel and practical methods for dynamic scheduling and resource management that can be integrated in production batch systems.

As the first contribution, this thesis presented a novel dynamic resource management method and a job scheduling algorithm to schedule unpredictably evolving jobs in a cluster environment. The implementation in the TORQUE/Maui batch system allows applications to request and release nodes during job execution without having to indicate in advance the evolving behavior at job submission time. The scheduling algorithm uses a fairness scheme that can be configured to control the fair allocation of resources between static and dynamic resource requests. The scheme is based on adjusting the delay caused to the queued jobs when resources are allocated to unpredictably arriving dynamic requests. Our evaluations with a modified ESP benchmark showed that scheduling these unpredictable dynamic requests has the potential to increase the system throughput, however, only at the cost of increasing the waiting time of queued jobs in an unfair manner. Using the fairness scheme, we showed that it is possible to control the delays caused to the queued jobs while also improving resource utilization and throughput.

As the second contribution, the thesis proposed an efficient dynamic scheduling algorithm for scheduling malleable jobs. The proposed scheduling algorithm, called DBES, schedules malleable jobs with the goal of improving the overall throughput of the system. The algorithm selects malleable jobs for expansion by analyzing reservation dependencies in the job queue and strikes a good balance between backfilling and malleable expand/shrink operations through its two stage expansion process. Also, the TORQUE/Maui batch system and the Charm++ runtime were extended with a protocol for malleability interactions. Evaluation of the DBES algorithm with a modified ESP benchmark containing Charm++ applications as malleable jobs showed that the algorithm consistently delivers higher throughput and system utilization compared to other malleable job scheduling strategies for varying fractions of malleable jobs in the workload.

As the third contribution, the thesis introduced a novel dynamic node replacement algorithm to achieve better resiliency in cluster environments. The node replacement algorithm aims to replace failed nodes of a job with healthy nodes on-the-fly. The dynamic node replacement algorithm is implemented as a supplement to the main job scheduling algorithm and is triggered only in the event of node failure. It uses unique features of the different job types to find quick node replacements for the interrupted jobs. The algorithm can find replacement resources by shrinking other running malleable jobs and does an efficient selection of jobs to be shrunk by analyzing the reservation dependencies. It also tries to find replacement resources by restarting moldable jobs with a lower number of processors. Again, it makes an efficient selection of moldable jobs to be restarted using linear programming to compute the delay that the makespan is projected to suffer. The evaluation showed that the dynamic node replacement algorithm can greatly reduce the loss in throughput that is caused due to frequent node failures.

As the final contribution, the thesis showed how dynamic resource management methods can naturally support usage scenarios of heterogeneous architectures with network-attached accelerators. This was exemplified with the DEEP and DAC architectures, where accelerators can be dynamically added to/released from jobs according to the needs of the application. This can not only enable applications to use as many accelerators as needed for different phases of its execution, but can also improve the overall accelerator utilization. We showed through evaluations that accelerators can be dynamically (de)allocated with negligible overhead.

The results of all the contributions have clearly shown that dynamic scheduling and resource management is a beneficial property in HPC clusters providing both user-level and system-level advantages. The methods proposed in this thesis are one of the first works towards providing a practical solution for enabling adaptivity in HPC clusters. They are also easily extendible with further useful features. For example, the DBES algorithm can be extended with features such as (i) expanding/shrinking malleable jobs with topology awareness, (ii) scheduling malleable jobs based on feedback from the application on its scaling patterns, and (iii) distributing idle resources to malleable jobs with fairness considerations.

A major impact of this thesis has been its strong influence in the creation of the ongoing effort to define a standard Scheduler and Malleable/Evolvable Application Dialog (SMEAD) API [141]. We were contacted by Adaptive Computing after the company became interested in the results of our work and had been wanting to create a standard API for adaptivity interactions between applications and batch systems. Such a standard API will make applications that use dynamic resource management facilities portable across cluster platforms. Also, it will ease the development of dynamic resource management features in batch systems and runtime systems of programming paradigms. Working together with Adaptive Computing and the PMIx development community [142], we furthered the ideas to create the standard API and announced the effort along with invitation for collaborations through a Birds-of-a-Feather session at the Supercomputing '15 Conference [143]. An open-ended survey was also released to collect information on adaptivity properties from developers of applications, programming models, and batch systems [144]. The API is being defined as part of the PMIx project with inputs from the survey and a number of partner organizations [145].

This work not only progresses the state of the art in this area but also opens a wide scope for further research. As already mentioned, the need for dynamic resource management and scheduling has never been greater than now in HPC because of several challenges in exploring new architectures and application domains. For example, energy efficiency is a key issue emerging out of increasing system size. This is being addressed in many ways such as developing energy efficient hardware, operating cluster systems with policies for energy efficiency, and optimizing software to make better use of the underlying hardware. In this regard, adaptive scheduling can be used to dynamically modify the resource allocations of jobs to optimize power consumption. The batch system could be tightly integrated with system monitoring tools, which will allow it to perform both static and dynamic scheduling based on power and energy data.

In similar lines, high resiliency is an indispensable requirement for upcoming exascale systems as the estimated high failure rates pose a serious threat to the robustness of a system. Although this thesis presented the dynamic node replacement algorithm as a solution to improving system availability, it can be extended with several useful features to further enhance system robustness. For instance, when the batch system is coupled with temperature monitoring tools and failure prediction mechanisms, it can be enriched to perform proactive migration automatically or at the command of the system administrator. This can potentially save applications from being interrupted by failures and facilitate a comparably smooth execution. This requires batch systems to be able to function together with checkpoint/restart frameworks and applications.

The data intensiveness of today's applications also provide an opportunity for optimizing resource utilization in the system through dynamic resource management. In classic cluster systems, a partition of nodes that are close to the parallel file system can be dynamically allocated to jobs performing I/O operations for buffering data and transferring them asynchronously to the file system. This can reduce the amount of time spent by an application on I/O operations.

Overall, it can be observed that dynamic scheduling and resource management not only provides scheduling benefits but also has the potential to support other aspects such as enabling faster I/O, improving fault tolerance and facilitating cost-efficient operation of a cluster system. Therefore, an integrated middleware that tightly couples adaptivity-capable batch systems, checkpoint/restart frameworks, monitoring tools and application runtimes is most desirable for future HPC systems.

Bibliography

[1] Norbert Eicker, Thomas Lippert, Thomas Moschny, and Estela Suarez. The DEEP Project An alternative approach to heterogeneous cluster-computing in the many-core era. *Concurrency and Computation: Practice and Experience*, 2015.

[2] Sebastian Rinke, Daniel Becker, Thomas Lippert, Suraj Prabhakaran, Lidia Westphal, and Felix Wolf. A Dynamic Accelerator-Cluster Architecture. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 357–366. IEEE Computer Society, 2012.

[3] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[4] H. Fröning, M. Nüssle, H. Leitz, C. Leber, and U. Brüning. On Achieving High Message Rates. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013.

[5] Top500 Computer Sites. http://top500.org, Nov 2015. Accessed: 2016-01-29.

[6] Introduction to Intel Xeon Phi Coprocessor. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, Nov 2012. Accessed: 2016-01-29.

[7] Tesla K20x GPU Accelerator. http://www.nvidia.de/content/PDF/kepler/Tesla-K20X-BD-06397-001-v07.pdf, Jul 2013. Accessed: 2016-01-29.

[8] Message Passing Interface (MPI) Forum. http://www.mpi-forum.org/. Accessed: 2016-01-29.

[9] The OpenMP API Specifications. http://openmp.org/wp/openmp-specifications/. Accessed: 2016-01-29.

[10] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*. ACM Press, September 1993.

[11] Laxmikant V. Kalé, Abhinav Bhatele, Eric J. Bohm, and James C. Phillips. NAnoscale Molecular Dynamics (NAMD). In D. Padua, editor, *Encyclopedia of Parallel Computing*. Springer Verlag, 2011.

[12] Nikhil Jain, Eric Bohm, Eric Mikida, Subhasish Mandal, Minjung Kim, Prateek Jindal, Qi Li, Sohrab Ismail-Beigi, Glenn Martyna, and Laxmikant Kalé. Openatom: Scalable ab-initio molecular dynamics with diverse capabilities. In *International Supercomputing Conference*, ISC HPC '16, 2016.

[13] P. Miller, M. Robson, B. El-Masri, R. Barman, G. Zheng, A. Jain, and L. Kalé. Scaling the isam land surface model through parallelization of inter-component data transfer. In *2014 43rd International Conference on Parallel Processing*, pages 422–431, Sept 2014.

[14] Jae-Seung Yeom, Abhinav Bhatele, Keith R. Bisset, Eric Bohm, Abhishek Gupta, Laxmikant V. Kalé, Madhav Marathe, Dimitrios S. Nikolopoulos, Martin Schulz, and Lukasz Wesolowski. Overcoming the scalability challenges of epidemic simulations on blue waters. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '14. IEEE Computer Society, May 2014.

[15] Nvidia Corp. CUDA Parallel Programming Platform. http://www.nvidia.com/object/cuda_home_new.html. Accessed: 2016-01-29.

[16] Khronos Group. Khronos OpenCL Registry - OpenCL Specifications. https://www.khronos.org/registry/cl/. Accessed: 2016-01-29.

[17] GNU. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/. Accessed: 2016-08-13.

[18] Rogue Wave Software. TotalView for HPC. http://www.roguewave.com/products-services/totalview. Accessed: 2016-08-13.

[19] Allinea. Allinea DDT: The debugger for C, C++ and F90 threaded and parallel code. http://www.allinea.com/products/ddt. Accessed: 2016-08-13.

[20] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[21] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, April 2010.

[22] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[23] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*, pages 1–12. ACM, November 2013.

[24] MPICH. High Performance Portable MPI. https://www.mpich.org/. Accessed: 2016-08-13.

[25] Open MPI. Open Source High Performance Computing. https://www.open-mpi.org/. Accessed: 2016-08-13.

[26] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, October 2003.

[27] BeeGFS. BeeGFS - The Parallel Cluster File System. http://www.beegfs.com/. Accessed: 2016-08-13.

[28] Lustre. Lustre Filesystem. http://lustre.org/. Accessed: 2016-08-13.

[29] OpenFabrics Software. OpenFabrics Enterprise Distribution. https://www.openfabrics.org/index.php/openfabrics-software.html. Accessed: 2016-08-13.

[30] Red Hat. Red Hat Enterprise Linux. https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux. Accessed: 2016-08-13.

[31] CentOS. The CentOS Project. https://www.centos.org/. Accessed: 2016-08-13.

[32] Dror G. Feitelson and Larry Rudolph. Towards Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996.

[33] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Task-Based Programming with OmpSs and Its Application. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 601–612. Springer International Publishing, 2014.

[34] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding Virtualization Capabilities to the Grid'5000 Testbed. In IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer, 2013.

[35] T.E. Carroll and D. Grosu. Incentive Compatible Online Scheduling of Malleable Parallel Jobs with Individual Deadlines. In *39th International Conference on Parallel Processing (ICPP)*, 2010.

[36] Hongyang Sun, Yangjie Cao, and Wen-Jing Hsu. Fair and Efficient Online Adaptive Scheduling for Multiple Sets of Parallel Applications. In *IEEE 17th International Conference on Parallel and Distributed Systems*, 2011.

[37] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, March 2011.

[38] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13):1175–1220, 2002.

[39] Wang Long, Lan Yuqing, and Xia Qingxin. Using CloudSim to Model and Simulate Cloud Computing Environment. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*, pages 323–328, Dec 2013.

[40] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[41] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson. Single Node On-Line Simulation of MPI Applications with SMPI. In *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 664–675, May 2011.

[42] David P. Bunde and Vitus J. Leung. PReMAS: Simulator for Resource Management. In *Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops*, ICPPW '14, 2014.

[43] K. Windisch, J.V. Miller, and V. Lo. ProcSimity: an experimental tool for processor allocation and scheduling in highly parallel systems. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the*, pages 414–421, Feb 1995.

[44] Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/. Accessed: 2016-01-29.

[45] Standard Workload Format. http://www.cs.huji.ac.il/labs/parallel/workload/swf.html. Accessed: 2016-01-29.

[46] Adrian T. Wong, Leonid Oliker, William T. C. Kramer, Teresa L. Kaltz, and David H. Bailey. ESP: A System Utilization Benchmark. In *Proc. of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000.

[47] S. Soner and C. Ozturan. Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 418–424, June 2012.

[48] Nicolas Capit, Georges Da, Costa Yiannis, Georgiou Guillaume Huard, Cyrille Martin, Grégory Mouniéand Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. In *In Cluster computing and Grid 2005 (CCGrid05*, pages 776–783, 2005.

[49] Yiannis Georgiou and Matthieu Hautreux. Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *Lecture Notes in Computer Science*, pages 134–156. Springer Berlin Heidelberg, 2013.

[50] Morris A. Jette, Andy B. Yoo, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer-Verlag, 2003.

[51] Moab hpc basic edition. http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/. Accessed: 2016-01-29.

[52] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.0, July 1997. available at: http://www.mpi-forum.org (Jul. 1997).

[53] Gladys Utrera, Julita Corbalan, and Jesus Labarta. Implementing Malleability on MPI Jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[54] K. El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. Malleable Iterative MPI Applications. *Concurrency and Computation: Practice and Experience*, 21, 2009.

[55] V.Gregory Weirs Tomasz Plewa, Timur Linde. Adaptive Mesh Refinement: Theory and Applications. Springer, 2003.

[56] Siegfried Müller. *Adaptive Multiscale Schemes for Conservation Laws*. Lecture Notes in Computational Science and Engg. Springer, 2003.

[57] F. Bramkamp, Ph. Lamby, and S. Müller. An Adaptive Multiscale Finite Volume Solver for Unsteady and Steady State Flow Computations. *J. Comput. Phys.*, July 2004.

[58] Preeti Malakar, Vijay Natarajan, Sathish S. Vadhiyar, and Ravi S. Nanjundiah. A Diffusion-Based Processor Reallocation Strategy for Tracking Multiple Dynamically Varying Weather Phenomena. In *42nd International Conference on Parallel Processing (ICPP)*, Oct 2013.

[59] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J Dongarra. A Proposal for User-Level Failure Mitigation in the MPI-3 Standard. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee, 2012.

[60] Jack Dongarra and et al. The International Exascale Software Project Roadmap. *International Journal on High Performance Computing Applications*, 2011.

[61] ETP4HPC. Strategic Research Agenda. http://www.etp4hpc.eu/strategy/strategic-research-agenda/. Accessed: 2016-01-29.

[62] Catello Di Martino, F Baccanico, W Kramer, J Fullop, Z Kalbarczyk, and R Iyer. Lessons Learned From the Analysis of System Failures at Petascale: The Case of Blue Waters. In *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

[63] N. Eicker, T. Lippert, T. Moschny, and E. Suarez. The DEEP Project - Pursuing Cluster-Computing in the Many-Core Era. In *42nd International Conference on Parallel Processing (ICPP)*, Oct 2013.

[64] TORQUE Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/. Accessed: 2016-01-29.

[65] David B. Jackson, Quinn Snell, and Mark J. Clement. Core Algorithms of the Maui Scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2001.

[66] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, Christian Windisch, and Felix Wolf. A Batch System with Fair Scheduling for Evolving Applications. In *Proc. of the 43rd International Conference on Parallel Processing (ICPP), Minneapolis, MN, USA*, September 2014.

[67] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kalé. A Batch System with Efficient Scheduling for Malleable and Evolving Applications. In *Proc. of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India*, pages 429–438. IEEE Computer Society, May 2015.

[68] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, and Felix Wolf. A Dynamic Resource Management System for Network-Attached Accelerator Clusters. In *Proc. of the 42nd International Conference on Parallel Processing (ICPP), Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS), Lyon, France*, pages 773–782, October 2013.

[69] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *2007 IEEE International Conference on Cluster Computing*, Sept 2007.

[70] Márcia C. Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe O. A. Navaux. Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*. Springer-Verlag, 2010.

[71] Boon-Ping Gan and Shell-Ying Huang. Scheduling dynamically evolving parallel programs using the genetic approach. In *The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, May 2000.

[72] S. Ghafoor, T. Haupt, I. Banicescu, R. Carino, and N. Ammari. A Resource Management System for Adaptive Parallel Applications in Cluster Environments. In *In Proceedings of the 6th International Conference on Linux Clusters: The HPC Revolution 2005*, April 2005.

[73] C. Klein and C. Perez. An RMS for Non-predictably Evolving Applications. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2011.

[74] D. Kumar, Zon-Yin Shae, and H. Jamjoom. Scheduling Batch and Heterogeneous Jobs with Runtime Elasticity in a Parallel Processing Environment. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012.

[75] Sathish S. Vadhiyar and Jack J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. In *In: Parallel Processing Letters. Volume*, 2002.

[76] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. A Framework for Dynamic Adaptation of Parallel Components. In *Proc. of the International Conference on Parallel Computing (ParCo) 2005*.

[77] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. A Malleable-Job System for Timeshared Parallel Machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2002.

[78] Gregory Mounie, Christophe Rapine, and Dennis Trystram. Efficient Approximation Algorithms for Scheduling Malleable Tasks. In *Proc. of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1999.

[79] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram. Approximation Algorithms for Scheduling Independent Malleable Tasks. In *Euro-Par 2001 Parallel Processing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001.

[80] J. Hungershofer. On the Combined Scheduling of Malleable and Rigid jobs. In *16th Symposium on Computer Architecture and High Performance Computing*, Oct 2004.

[81] Gladys Utrera, Siham Tabik, Julita Corbalan, and Jesus Labarta. A Job Scheduling Approach for Multi-core Clusters Based on Virtual Malleability. In *Euro-Par 2012 Parallel Processing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[82] Márcia C. Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe O. A. Navaux. Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*. Springer-Verlag, 2010.

[83] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *IEEE International Conference on Cluster Computing*, Sept 2007.

[84] Platform LSF. http://www-03.ibm.com/systems/platformcomputing/products/lsf/. Accessed: 2016-01-29.

[85] Walfredo Cirne and Francine Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002.

[86] Srividya Srinivasan, Vijay Subramani, Rajkumar Kettimuthu, Praveen Holenarsipur, and P. Sadayappan. Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. In *High Performance Computing (HiPC) 2002*, volume 2552 of *Lecture Notes in Computer Science*, pages 174–183. Springer Berlin Heidelberg, 2002.

[87] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling technology for moldable scheduling of parallel jobs. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 92–99, Dec 2003.

[88] Gerald Sabin, Matthew Lang, and P. Sadayappan. Moldable Parallel Job Scheduling Using Job Efficiency: An Iterative Approach. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 4376 of *Lecture Notes in Computer Science*, pages 94–114. Springer Berlin Heidelberg, 2007.

[89] Allen B Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.

[90] S. Asghar, E. Aubanel, and D. Bremner. A Dynamic Moldable Job Scheduling Based Parallel SAT Solver. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 110–119, Oct 2013.

[91] Kuo-Chan Huang, Tse-Chi Huang, Mu-Jung Tsai, and Hsi-Ya Chang. Moldable Job Scheduling for HPC as a Service. In James J. (Jong Hyuk) Park, Ivan Stojmenovic, Min Choi, and Fatos Xhafa, editors, *Future Information Technology*, volume 276 of *Lecture Notes in Electrical Engineering*, pages 43–48. Springer Berlin Heidelberg, 2014.

[92] Song Wu, Qiong Tuo, Hai Jin, Chuxiong Yan, and Qizheng Weng. HRF: A Resource Allocation Scheme for Moldable Jobs. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 17:1–17:8, New York, NY, USA, 2015. ACM.

[93] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for Linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[94] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 32. ACM, 2011.

[95] A. Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. of IEEE/ACM SC'10*, pages 1–11, 2010.

[96] HTCondor. http://research.cs.wisc.edu/htcondor/. Accessed: 2016-01-29.

[97] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, 2014.

[98] HwaMin Lee, DooSoon Park, Min Hong, Sang-Soo Yeo, SooKyun Kim, and SungHoon Kim. A Resource Management System for Fault Tolerance in Grid Computing. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 2, pages 609–614, Aug 2009.

[99] Garrick Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006.

[100] Robert L. Henderson. Job Scheduling Under the Portable Batch System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1995.

[101] Adaptive Computing, Inc. http://www.adaptivecomputing.com/. Accessed: 2016-01-29.

[102] Maui Administrator Guide. http://docs.adaptivecomputing.com/maui/mauiadmin.php. Accessed: 2016-01-29.

[103] D. R. Welch, T. C. Genoni, R. E. Clark, and D. V. Rose. Adaptive Particle Management in a Particle-in-cell Code. *J. Comput. Phys.*, 227(1):143–155, November 2007.

[104] Frank Dieter Bramkamp. *Unstructured h-Adaptive Finite-Volume Schemes for Compressible Viscous Fluid Flow*. dissertation, RWTH Aachen, 2003.

[105] Philipp Lamby. *Parametric Multi-Block Grid Generation and Application to Adaptive Flow*. dissertation, RWTH Aachen, 2003.

[106] Brix, K., Melian, S., Müller, S., and Bachmann, M. Adaptive Multiresolution Methods: Practical issues on Data Structures, Implementation and Parallelization*. *ESAIM: Proc.*, 34:151–183, 2011.

[107] D.F. Martin, P. Colella, M. Anghel, and F.J. Alexander. Adaptive mesh refinement for multiscale nonequilibrium physics. *Computing in Science Engineering*, May 2005.

[108] Greg L. Bryan, Tom Abel, and Michael L. Norman. Achieving Extreme Resolution in Numerical Cosmology Using Adaptive Mesh Refinement: Resolving Primordial Star Formation. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, New York, USA, 2001. ACM.

[109] Christian Windisch, Birgit Reinartz, and Siegfried Müller. *Numerical Simulation of Coolant Variation in Laminar Supersonic Film Cooling*. American Institute of Aeronautics and Astronautics, February 2012.

[110] Christian Windisch, B. Reinartz and Siegfried Müller. H-Adaptive Simulation of Hypersonic Flows in Thermochemical Nonequilibrium. American Institute of Aeronautics and Astronautics, 2012.

[111] Claudia Leopold, Michael Süß, and Jens Breitbart. Programming for malleability with hybrid MPI-2 and OpenMP: Experiences with a simulation program for global water prognosis. *Proceedings of the European Conference on Modelling and Simulation*, pages 665–670, 2006.

[112] Travis Desell, Kaoutar El Maghraoui, and Carlos A. Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.

[113] Jon Purnell, Malik Magdon-Ismail, and Heidi Jo Newberg. A Probabilistic Approach to Finding Geometric Objects in Spatial Datasets of the Milky Way. In *Foundations of Intelligent Systems*, volume 3488 of *Lecture Notes in Computer Science*, pages 485–493. Springer Berlin Heidelberg, 2005.

[114] K. El Maghraoui, T.J. Desell, B.K. Szymanski, and C.A. Varela. Dynamic Malleability in Iterative MPI Applications. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 591–598, May 2007.

[115] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive Scheduling with Parallelism Feedback. In *Proc. of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[116] Hongyang Sun, Yangjie Cao, and Wen-Jing Hsu. Efficient Adaptive Scheduling of Multiprocessors with Stable Parallelism Feedback. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[117] Filippo Gioachin, Chee Wai Lee, and Laxmikant V. Kalé. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid'09*, 2009.

[118] A. Bhatele, S. Kumar, Chao Mei, J.C. Phillips, Gengbin Zheng, and Laxmikant V. Kalé. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *IEEE International Symposium on Parallel and Distributed Processing, 2008*.

[119] Marcel Neumann. A Resilience-enabling Resource Manager for HPC. Master's thesis, RWTH Aachen University, Aachen Germany, 2015.

[120] Gengbin Zheng, Xiang Ni, and L.V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.

[121] A. Rezaei and F. Mueller. Sustained resilience via live process cloning. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1498–1507, May 2013.

[122] The University of Edinburgh. SimJava2.0. http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/. Accessed: 2016-01-29.

[123] GNU project. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/. Accessed: 2016-01-29.

[124] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[125] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference*, pages 483–485. ACM, 1967.

[126] Kenneth C Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *ACM SIGMETRICS Performance Evaluation Review*, volume 17, pages 171–180, 1989.

[127] Bahman Javadi, Derrick Kondo, Alexandru Iosup, and Dick Epema. The Failure Trace Archive: Enabling the comparison of failure measurements and models of distributed systems. *Journal of Parallel and Distributed Computing*, 73(8):1208–1223, 2013.

[128] Bianca Schroeder, Garth Gibson, et al. A large-scale study of failures in high-performance computing systems. *Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[129] Franck Cappello, Al Geist, William Gropp, Sanjay Kalé, Bill Kramer, and Marc Snir. Toward Exascale Resilience: 2014 Update. 2014.

[130] The DEEP Project. http://www.deep-project.eu/.

[131] Sebastian Rinke, Suraj Prabhakaran, and Felix Wolf. Efficient Offloading of Parallel Kernels Using MPI_Comm_spawn. In *Proc. of the 42nd International Conference on Parallel Processing (ICPP), Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA), Lyon, France*, pages 877–884, October 2013.

[132] ParaStation ClusterTools. http://www.par-tec.com/products/parastation-clustertools.html. Accessed: 2016-01-29.

[133] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel; Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, 2015.

[134] Infiniband Trade Association. Infiniband Roadmap. http://www.infinibandta.org/content/pages.php?pg=technology_overview. Accessed: 2016-08-13.

[135] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10. IEEE Computer Society, 2011.

[136] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2009.

[137] Amnon Barak, Tal Ben-Nun, Ely Levy, and Amnon Shiloh. A package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In *Proc. of the International Conference on Cluster Computing (PPAAC Workshop)*. IEEE, September 2010.

[138] SanDisk. Fusion ioMemory Technology. https://www.sandisk.com/business/datacenter/products. Accessed: 2016-08-13.

[139] A. Foong and F. Hady. Storage As Fast As Rest of the System. In *2016 IEEE 8th International Memory Workshop (IMW)*, pages 1–4, 2016.

[140] Intel. Non-Volatile Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html. Accessed: 2016-08-13.

[141] Adaptive Computing. Job Schedulers and Malleable/Evolving Jobs. http://www.adaptivecomputing.com/blog-hpc/job-schedulers-and-malleableevolving-jobs-part-4/. Accessed: 2016-09-08.

[142] Open MPI. PMI Exascale (PMIx). https://www.open-mpi.org/projects/pmix/. Accessed: 2016-09-08.

[143] Supercomputing '15. Charting the PMIx Roadmap. http://sc15.supercomputing.org/schedule/event_detail-evid=bof122.html. Accessed: 2016-09-08.

[144] Adaptive Computing. Malleable/Evolvable Application/RTE Use Cases. https://docs.google.com/forms/d/e/1FAIpQLScJl4RDaip6c81ve1hgOnF1A5LeVRt4PjqjeaLstSwz-mgN8w/viewform?c=0&w=1. Accessed: 2016-09-08.

[145] PMIx Programmers Manual. PMI-Exascale. http://pmix.github.io/master/. Accessed: 2016-09-08.